MASTER EN SCIENCES INFORMATIQUES

Contribution à la mise en oeuvre d'un logiciel X400 : utilisation des langages formels de spécification et réalisation d'un outil d'analyse et de production d'unités de données de protocole

Brossel, Philippe

Award date:
1987

Awarding institution:
Universite de Namur

Facultés Universitaires
Notre-Dame de la Paix, Namur
Année académique 1986-1987

Contribution à la mise en oeuvre d'un logiciel X400 :

Utilisation des langages formels de spécification et réalisation d'un outil d'analyse et de production d'unités de données de protocole

Philippe Brossel

Mémoire présenté en vue de
l'obtention du titre de
licencié et maître en informatique

Je tiens à exprimer ma profonde reconnaissance à Monsieur Van Bastelaer, qui a accepté d'assurer la direction de ce mémoire, pour le temps et l'attention qu'il m'a consacrés.

J'adresse mes plus vifs remerciements à Monsieur Gillet de la firme Electronique et Télécommunication Bell, ainsi qu'à tous les membres du personnel, pour leur accueil chaleureux et l'attention qu'ils m'ont témoigné durant le stage. Que Messieurs Paris et Fisette se trouvent ici tout particulièrement assurés de ma profonde gratitude pour leur soutien constant et leurs conseils judicieux.

Mes remerciements vont également à Monsieur Celiktin pour son aimable disponibilité et ses remarques pertinentes sur une version préliminaire du présent travail.

Enfin, je voudrais remercier tous ceux qui, professeurs, parents ou amis ont contribué, à quelque titre que ce soit, à ma formation et à la réalisation de ce mémoire.

Table des matières

3. Les langages formels de spécification

4. Outil d'analyse et de production d'unités de données de protocole

5. Réalisation de l'outil d'analyse et de production d'unités de données de protocole

Introduction

L'informatique répartie, constituée de systèmes informatiques distribués, tend à prendre une place de plus en plus importante. Elle met en relation des installations indépendantes et des équipements hétérogènes - fournis par différents constructeurs - qui coopèrent en vue de réaliser une tâche commune distribuée.

Les modalités d'interaction entre ces éléments sont exprimées par le biais de règles, appelées protocoles, que chacun de ceux-ci doit respecter pour permettre un dialogue effectif. Les protocoles font l'objet d'une normalisation au niveau international et sont proposés par les organismes de standardisation comme référence pour les fournisseurs de matériel et de logiciel informatique.

Il est important que ces protocoles soient énoncés de manière claire et précise, sans quoi ils risquent d'être interprétés différemment par les constructeurs lors de l'implémentation.

Les textes normalisés sont rédigés en langue naturelle et autorisent par le fait même des interprétations différentes, voire opposées.

Des implémentations logicielles et matérielles se référant à ces propositions de protocoles, et réalisées indépendamment les unes des autres, risquent donc de ne pas pouvoir s'interconnecter valablement. En effet, les interprétations données aux textes normalisés imposent certains choix de mise en oeuvre, qui rendent difficile la communication entre des implémentations issues de vues différentes de la même proposition.

Pour réduire le risque d'interprétations divergentes, les organismes de standardisation envisagent de compléter les descriptions de protocoles par des spécifications rédigées dans un langage formel. Elles décrivent les règles de dialogue entre composants distribués et l'aspect des informations qu'ils peuvent s'échanger.

Etant donné que de tels langages interviennent seulement dans un domaine d'application très spécifique, il n'est pas nécessaire que leur formalisme soit à même d'exprimer n'importe quelle classe de problèmes. Celui-ci peut alors être défini par un nombre restreint de règles très précises. L'information contenue dans un texte formel est extraite suivant les conventions ainsi énoncées et ne peut donner lieu qu'à une seule interprétation.

D'autre part, un texte de spécification rédigé dans un langage formel peut faire l'objet d'un traitement automatisé, étant donné que l'extraction de l'information qu'il contient s'effectue à l'aide des règles précises relatives au formalisme utilisé. On peut donc envisager la conception et l'utilisation d'outils de haut niveau, intervenant dans les phases de développement et de mise au point d'un logiciel distribué.

Nous présenterons dans ce travail la réalisation d'un outil intervenant dans le cadre d'un langage de spécification de structures de donnée, proposé par l'organisme de standardisation CCITT (Comité Consultatif International pour la Télégraphie et la Téléphonie). Il permet l'analyse et la production des informations échangées entre des applications distribuées, appelées unités de donnée de protocole. Les caractéristiques de ces données sont spécifiées à l'aide du langage formel proposé par le CCITT.

Dans le chapitre 1, nous rappellerons brièvement les concepts de base relatifs à l'expression des règles de dialogue entre composants distribués.

Le chapitre 2 est consacré à la présentation d'une application distribuée de messagerie éléctronique. Elle fait l'objet d'une normalisation au niveau international, et constitue un domaine potentiel d'application de l'outil proposé.

Nous présenterons au chapitre 3 quelques langages formels de spécification, proposés par les organismes de standardisation.

Nous exposerons au chapitre 4 les fonctionnalités offertes par l'outil d'analyse et de production d'unités de donnée de protocole, en mettant en évidence les domaines potentiels d'application.

La mise en oeuvre d'un tel outil dans le cadre d'une spécification formelle particulière est présentée au chapitre 5. Nous pourrons ainsi mettre en évidence un ensemble de règles permettant de déduire automatiquement l'outil d'un texte de spécification quelconque.

Le chapitre 6 expose la conception et la réalisation d'un générateur automatique d'outils d'analyse et de production exploitant les informations contenues dans une spécification quelconque pour adapter l'outil proposé.

1. Interconnexion des systèmes ouverts

1.1. Introduction

Dans le but d'interconnecter des systèmes informatiques fournis par différents constructeurs, l'organisme international de standardisation ISO a défini un ensemble de standards dont le respect strict doit rendre possible l'échange d'informations entre de tels systèmes. Ces standards constituent le modèle de référence pour l'interconnexion des systèmes ouverts, appelé modèle OSI (Open Systems Interconnection) [9].

Un système informatique peut être qualifié d'ouvert lorsqu'il respecte ces standards et permet ainsi à d'autres systèmes d'entrer en communication avec lui. Ces systèmes ouverts peuvent donc échanger les informations nécessaires pour mener à bien des tâches communes distribuées. Le respect des standards ISO permet ces échanges sans qu'il soit pour autant nécessaire que les deux systèmes utilisent des matériels similaires ou qu'ils reposent sur des implémentations semblables. Seul le comportement externe de chacun des systèmes doit être celui imposé par les normes. La description des données qu'un tel système échange avec son environnement et de ses réactions lors de la communication suffit pour exprimer le comportement du système.

Le modèle OSI est également appelé "modèle des sept couches". Nous allons dans ce qui suit en analyser les principales caractéristiques.

1.2. Concepts de base d'un modèle en couches

Etant donné qu'un système doit respecter les standards ISO pour être qualifié d'ouvert, il est nécessaire que ces exigences soient énoncées de manière telle que l'on puisse aisément affirmer qu'un système est ouvert ou non. Il s'agit en effet d'énoncer clairement les contraintes auxquelles les implémentations doivent satisfaire. Les standards ne doivent pas non plus imposer trop de restrictions quant à l'implémentation elle-même.

Il est donc nécessaire de disposer d'un moyen permettant de spécifier de manière exacte comment un système doit se comporter pour être ouvert. De plus, ces spécifications doivent se limiter à la description du comportement externe, pour garantir la liberté des constructeurs quant à l'implémentation.

La spécification de ce comportement est un problème très complexe. C'est pourquoi on le subdivise en sous-problèmes plus spécifiques et plus restreints. Ceux-ci ont l'avantage d'être plus dominables. La spécification des différents sous-systèmes, combinée à la description de l'agencement des différents composants, permet alors de spécifier le système complet.

Le système ouvert dont on désire spécifier le comportement est donc subdivisé en un ensemble de sous-systèmes. Les sous-systèmes de même rang constituent une couche du modèle complet, appelée couche(N). Chaque couche(N) regroupe une série d'entités (entités(N)) fournissant toutes aux utilisateurs de cette couche un même service, dénommé service(N).

Chaque couche du modèle ne connait par définition que les deux couches adjacentes. Elle peut rendre le service(N) en utilisant le service(N-1) et des fonctionnalités propres. Une couche(N) accède au service(N-1) par l'intermédiaire d'un SAP (Service Access Point) et échange avec la couche inférieure des éléments de données appelés SDU (Service Data Unit).

Les entités appartenant aux couches(N) de deux systèmes ouverts communiquants sont appelées entités paires. Le dialogue entre celles-ci est régie par une spécification dénommée protocole, qui définit exactement le comportement externe attendu. Les éléments de données échangés entre ces entités paires sont appelées unités de données de protocole ou PDUs (Protocol Data Units).

Ces définitions sont représentées schématiquement à la figure 1.1.

La division du système à spécifier en couches indépendantes possède un autre avantage. La couche(N) peut en effet faire abstraction des détails d'implémentation de toutes les autres couches, étant donné qu'elle ne doit connaître du reste du système que ses interfaces avec les deux couches adjacentes.

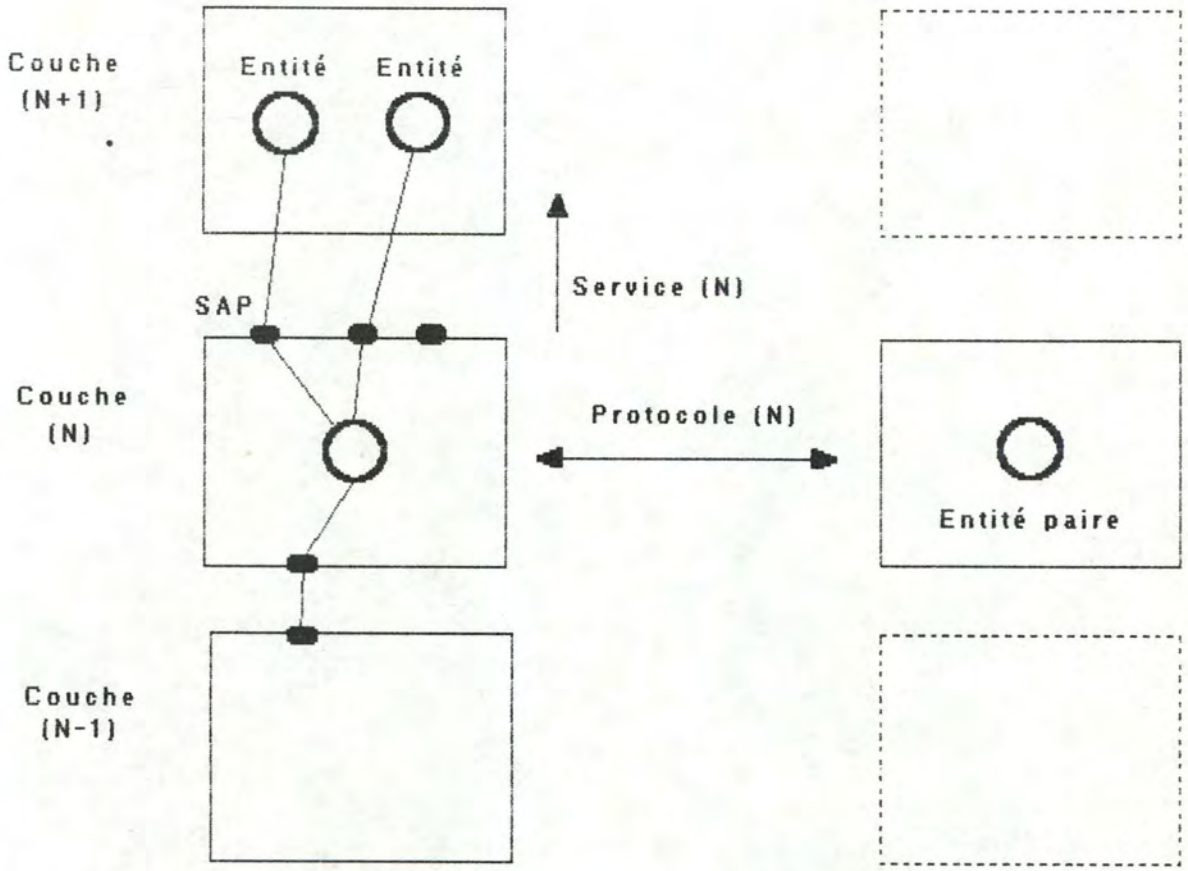


Figure 1.1

Principes de structuration
en couches

1.3. Présentation rapide du modèle de référence

Le modèle de référence pour l'interconnexion des systèmes ouverts est représenté à la figure 1.2. Il est composé de sept couches, dont nous allons brièvement exposer les fonctionnalités. Nous accorderons plus d'importance aux couches 6 et 7, étant donné que les chapitres suivants traitent essentiellement de ces couches.

On peut distinguer, dans le modèle OSI, deux modes de dialogue entre entités paires. Les entités peuvent dialoguer suivant un mode "orienté connexion" (connection oriented), ou selon un mode "sans connexion" (connection-less).

On distingue dans le premier mode de dialogue trois phases : la première est destinée à établir explicitement une communication entre les deux entités, la seconde phase est

destinée à l'échange des informations, et la dernière est la clôture de la communication. Une véritable voie de communication est ainsi établie avant l'échange, et il n'est donc plus nécessaire d'ajouter aux données "utiles", des informations permettant de désigner l'entité à laquelle on s'adresse. Ce mode de dialogue se justifie lorsqu'un volume assez important de données doit être échangé.

Le second mode de dialogue est dit "sans connexion", étant donné qu'il n'existe pas de phase d'initialisation et de clôture de la communication. Les informations à échanger sont émises de manière asynchrone, et chaque entité doit s'attendre à recevoir à n'importe quel moment des informations de son entité paire. Ce mode est généralement utilisé lorsqu'il est nécessaire d'échanger de temps en temps des informations. En effet, on est contraint d'ajouter aux données "utiles", des informations permettant de désigner l'entité à laquelle sont destinées ces données.

Ces deux modes de dialogue n'existent cependant que dans les couches supérieures (couches réseau, transport, session, présentation et application). Pour les couches inférieures (couches physique et de liaison de données), on ne retient que le mode "orienté connexion".

Les définitions qui suivent s'inspirent largement des références [1], [2] et [4].

1.3.1. La couche physique

La couche physique offre les procédures et moyens matériels permettant l'établissement, le maintien et la libération d'une communication physique entre deux composants de liaison. Cette liaison est utilisée par la couche 2 du modèle de référence. Le service qu'elle offre est une transmission de bits.

1.3.2. La couche de liaison de données

La couche de liaison de données fournit les procédures et moyens fonctionnels nécessaires à l'établissement, le maintien et la libération d'une liaison de données entre deux composants de la couche réseau. La couche 2 peut en plus assurer un certain degré de correction des données transmises. Les données transmises peuvent être quelconques.

1.3.3. La couche réseau

La couche réseau permet aux entités transport d'établir une communication avec une entité paire distante. La transmission d'information s'effectue via un certain nombre de noeuds intermédiaires. Cette couche offre un service de bout en bout.

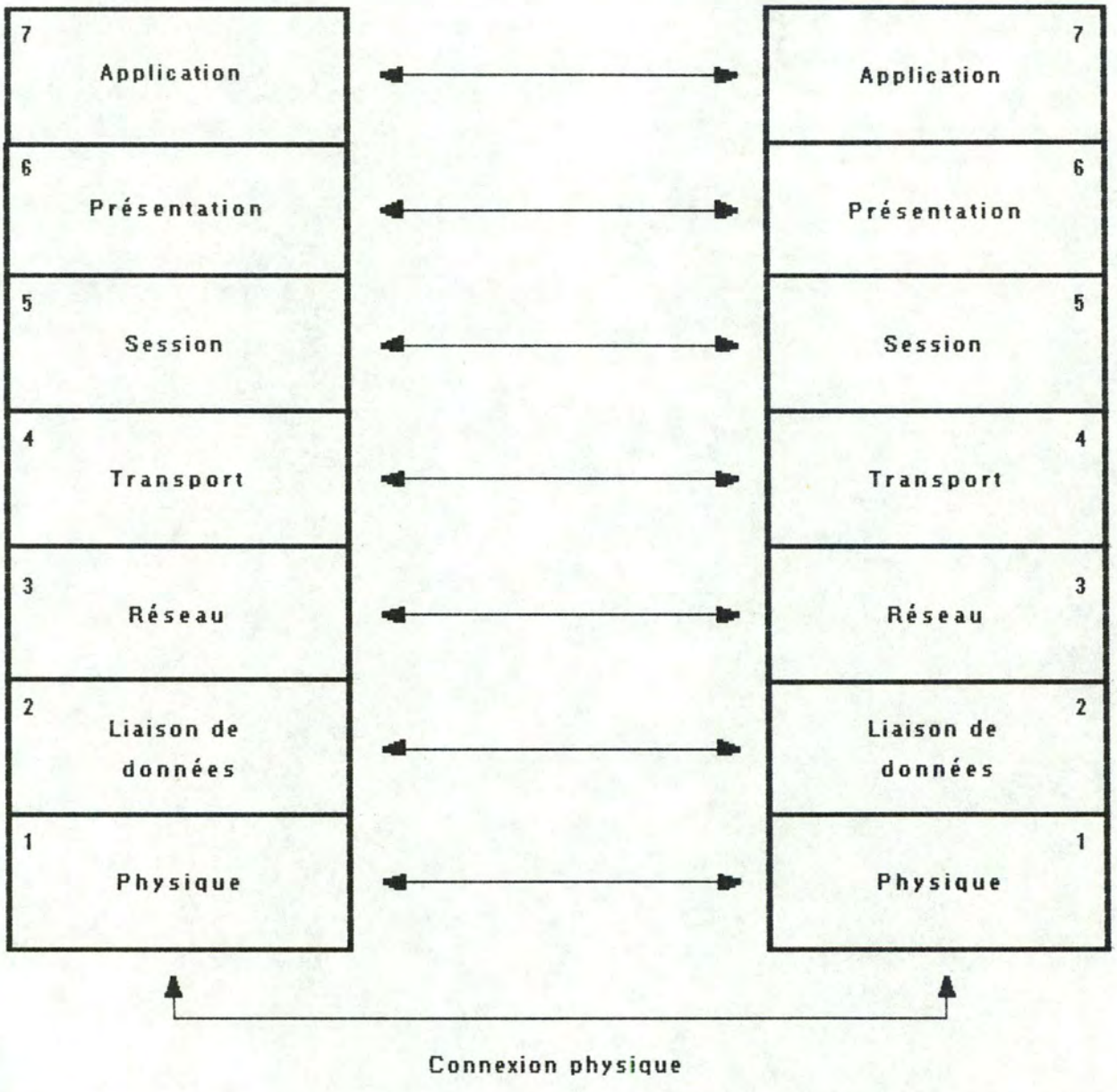


Figure 1.2

Le modèle de référence OSI

1.3.4. La couche transport

La couche transport permet essentiellement l'expression de contraintes de performance. Celles-ci indiquent par exemple la vitesse de transmission désirée, ou le degré de fiabilité de la communication, tant au niveau de la disponibilité de la communication que du point de vue de la correction des données transmises.

Cette couche peut donc offrir aux entités session un service plus ou moins élaboré de transport de données.

1.3.5. La couche session

La couche session introduit la notion de décomposition logique d'une communication. Elle permet de diviser un dialogue en unités logiques (activités), pouvant être mémorisées, interrompues, reprises, ... Cette couche permet aux entités présentation de disposer d'un véritable service de session, tel qu'il peut exister lors du dialogue d'un utilisateur humain avec une machine.

1.3.6. La couche présentation

La couche présentation résout le problème de l'incompatibilité des formats de codage des informations. En effet, les applications désirent être indépendantes du format des données manipulées par les applications distantes auxquelles elles s'adressent, tant au niveau de la représentation utilisée pour le stockage local que pour le format intervenant dans l'échange.

La spécification du modèle de référence autorise les entités de chaque couche à utiliser un format propre pour le stockage local des informations et l'échange de données avec les couches adjacentes. Il n'en est pas de même pour le format des données échangées entre les entités paires. Pour les couches inférieures, ces formats sont fixés "au bit près" de manière standardisée. Ainsi toute entité utilise le même format de transfert que l'entité paire.

Il n'en est pas de même pour les entités application. Elles ont en effet le droit d'émettre des informations codées suivant un format qui leur est propre. Ceci pose bien entendu un problème de "compréhension mutuelle" des entités paires.

C'est pourquoi les entités paires de la couche présentation négocient, lors d'une demande de connexion entre entités application, un format commun de données, appelé syntaxe concrète, qui permet de transférer correctement les informations. Chacune des entités présentation fournit alors à l'application qu'elle dessert les données reçues, en opérant les transformations éventuelles de formats. L'entité application peut dès lors se comporter "comme si" l'entité paire utilisait le même format de codage qu'elle lors de la communication.

L'ensemble des syntaxes concrètes qu'une entité présentation peut gérer est évidemment dépendant du contexte dans lequel travaille l'application. A chaque contexte application seront donc associées un ensemble de syntaxes concrètes.

Il faut également que les entités présentation paires puissent s'accorder sur la syntaxe concrète à utiliser lors de la communication. Dans le cas où il n'existe pas de point commun, le dialogue entre les entités application paires est impossible.

1.3.7. La couche application

La couche application englobe toutes les portions de programmes d'application susceptibles d'entrer en communication avec des programmes d'application éloignés.

Nous ne considérerons ici que ces portions de programme, que nous appellerons simplement applications. Les utilisateurs possibles de ces applications sont soit un programme quelconque, soit un utilisateur humain.

Ces applications sont totalement indépendantes des entités paires quant au format de représentation des données échangées. Nous avons vu que les entités présentation effectuent les transformations nécessaires, de manière à ce que chaque entité application puisse supposer que l'entité paire utilise lors de l'échange le même format.

Il est cependant nécessaire de pouvoir spécifier exactement quelles données les entités application peuvent échanger et leur signification. Mais cette spécification doit être indépendante des représentations choisies effectivement par les applications.

On exprime donc la sémantique des données échangées plutôt que leur format réel, en utilisant un formalisme

abstrait. Il doit être possible de représenter une valeur exprimée par le biais de ce formalisme, dans le format choisi par chacune des entités.

Ce formalisme est une syntaxe abstraite, et permet de décrire la sémantique des données échangées, sans pour autant faire explicitement référence à une représentation concrète particulière.

Une syntaxe abstraite est constituée d'une part d'une dénomination des types de donnée, et d'autre part d'une désignation des valeurs que ces informations peuvent prendre. On spécifie donc les données échangées entre les applications paires non pas en utilisant leur format de représentation réel, mais une désignation abstraite. On peut ainsi spécifier les éléments de données échangés, et le comportement des entités sans devoir faire explicitement référence à un format quelconque choisi par une implémentation.

Dans l'exemple de la figure 1.3, les entités application A1 et A2 désirent échanger des chaînes de caractères. Elles travaillent dans un contexte de transfert de fichiers de texte.

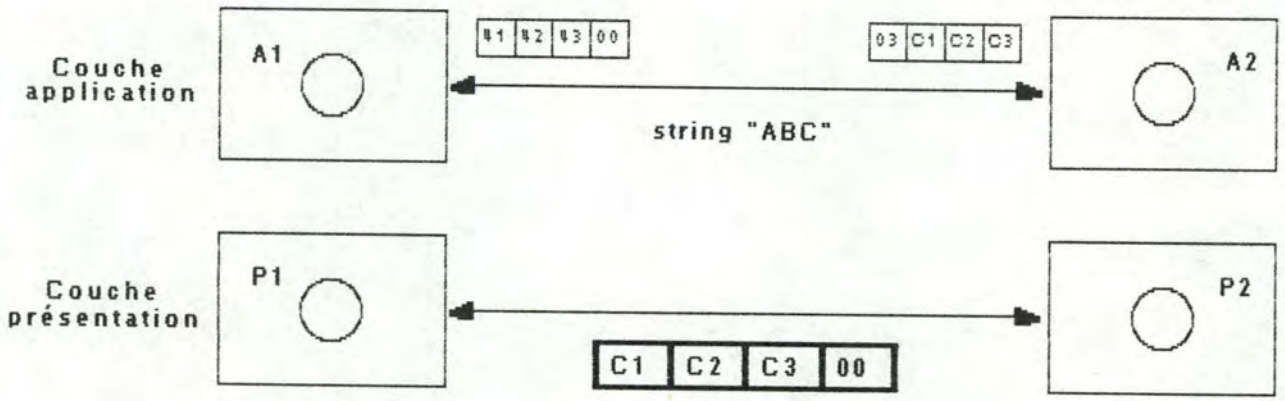


Figure 1.3

Exemple d'utilisation de syntaxes abstraite et concrète

L'application A1 utilise pour l'échange le format suivant : les caractères sont codés en ASCII, la chaîne est représentée par la concaténation de ces codes, et terminée par un code spécial $(00)_{16}$.

L'application A2 utilise un autre format : les caractères sont représentés en EBCDIC, la chaîne est

constituée de ces codes, et précédée d'un octet indiquant le nombre de caractères codés.

On ne peut exprimer les caractéristiques sémantiques des données échangées entre A1 et A2 que par l'intermédiaire d'une syntaxe abstraite. Elle permet de définir le type des données, et de dénoter les valeurs correspondantes, sans faire référence aux codages utilisés par les deux applications lors de l'échange. On définit par exemple le type string (chaîne de caractères), et on dénote alors les valeurs par la suite des caractères entourée par des guillemets. On note par exemple la valeur "ABC" du type string pour désigner la chaîne constituée des caractères A, B et C. Chaque entité application représente différemment la même valeur abstraite.

Ainsi A1 émet pour la valeur abstraite "ABC" du type string, une suite de quatre octets : $(41\ 42\ 43\ 00)_{16}$. Le premier octet contient la code ASCII du caractère A, le second celui de B, et le troisième est la code ASCII du caractère C. Le dernier octet indique la fin de la chaîne.

L'entité paire A2 reçoit la même information, mais codée différemment : $(03\ C1\ C2\ C3)_{16}$. Le premier octet indique que la chaîne est constituée de trois caractères, dont les codes EBCDIC sont donnés par les trois octets suivants.

Pour permettre le dialogue entre A1 et A2, les entités présentation P1 et P2 négocient une représentation commune à utiliser lors de l'échange (syntaxe concrète). Elles s'accordent par exemple sur la syntaxe suivante : les chaînes de caractères sont représentées par la suite des codes EBCDIC correspondant aux caractères, terminée par le code spécial $(00)_{16}$.

Lorsque l'entité application A1 émet une chaîne de caractères, codée suivant ses conventions propres, l'entité présentation P1 transforme la valeur fournie de manière à respecter la syntaxe concrète. Lors de la réception de cette valeur, l'entité paire P2 effectue les transformations adéquates pour fournir l'information à l'entité A2 dans le format qu'elle utilise.

Il faut noter que la syntaxe abstraite n'est utilisée que pour décrire les aspects sémantiques de l'information échangée entre les entités application. Elle est nécessaire étant donné qu'il n'est pas possible d'exprimer de manière concrète cette information.

2. Messagerie électronique X400

2.1. Introduction

Le modèle de référence pour l'interconnexion des systèmes ouverts, abordé au chapitre 1, est destiné à réduire l'écart entre des systèmes informatiques incompatibles. Il rend possible un dialogue constructif entre des applications distribuées, en permettant à chacune une relative indépendance par rapport à l'application à laquelle elle s'adresse.

Ces logiciels distribués peuvent être des applications développées et utilisées au sein d'une entreprise. Il existe également des applications "d'utilité générale", destinées à servir à un grand nombre d'utilisateurs n'appartenant pas nécessairement à la même organisation ou à la même entreprise.

Une de ces applications est la messagerie électronique. Elle touche les domaines de la communication interpersonnelle (à caractère privé ou professionnel) tout aussi bien que la communication entre programmes d'application s'exécutant sur une ou plusieurs machines.

Un système de messagerie est comparable au système postal. Il se distingue cependant par la multitude des services qu'il peut offrir. Etant donné que les moyens mis en oeuvre sont électroniques, on assure ces services avec rapidité et efficacité.

Des systèmes de messagerie sont déjà disponibles sur un grand nombre de machines, mais ne permettent la communication qu'entre des utilisateurs connectés au même ordinateur. Dans ce cas-ci, il n'existe pas de problème d'interconnexion ou d'incompatibilité, étant donné que le logiciel a été réalisé par un seul constructeur, et qu'il ne prévoit pas de communication avec un autre système de messagerie.

Un système de messagerie n'est cependant vraiment utile que lorsqu'il permet à un grand nombre d'utilisateurs de communiquer. Ceci implique qu'il est nécessaire d'interconnecter les systèmes existants, par l'intermédiaire de réseaux de communication, de manière à pouvoir mettre en relation les différentes communautés d'utilisateurs.

Dans ce contexte, on constate l'importance cruciale d'une compatibilité entre de tels systèmes. Celle-ci peut être atteinte par l'utilisation d'une architecture de communication OSI. Il est également nécessaire de spécifier exactement les fonctionnalités que ces systèmes doivent assumer pour permettre une collaboration effective.

L'organisme international de standardisation CCITT a proposé en 1984 un ensemble de recommandations visant à définir exactement ces fonctionnalités. Nous donnerons au point 2.2 une présentation générale de ces propositions.

Nous aborderons au point 2.3 quelques services intéressants pouvant être disponibles aux utilisateurs d'un système de messagerie.

Le point 2.4 présente brièvement un projet de recherche visant à utiliser les services d'une messagerie pour l'échange de documents multimédia.

2.2. Définitions

Un système de messagerie électronique (figure 2.1) est un ensemble de moyens matériels et logiciels permettant à ses utilisateurs d'échanger des messages de types variés. Les définitions qui suivent sont tirées de [8], [19], [20] et [21].

Les utilisateurs d'un tel système peuvent être aussi bien des programmes d'application que des êtres humains. Ils assument le rôle d'émetteurs (originator) lorsqu'ils envoient des messages à un ou plusieurs utilisateurs. Ces derniers assument alors le rôle de récepteurs (recipients).

On distingue donc deux types d'interactions entre un système de messagerie et ses utilisateurs : l'émission et la réception de messages. Un utilisateur émet un message lorsqu'il désire le faire parvenir à un récepteur. Cet utilisateur reçoit alors le message, dès que le système l'a acheminé jusqu'à lui.

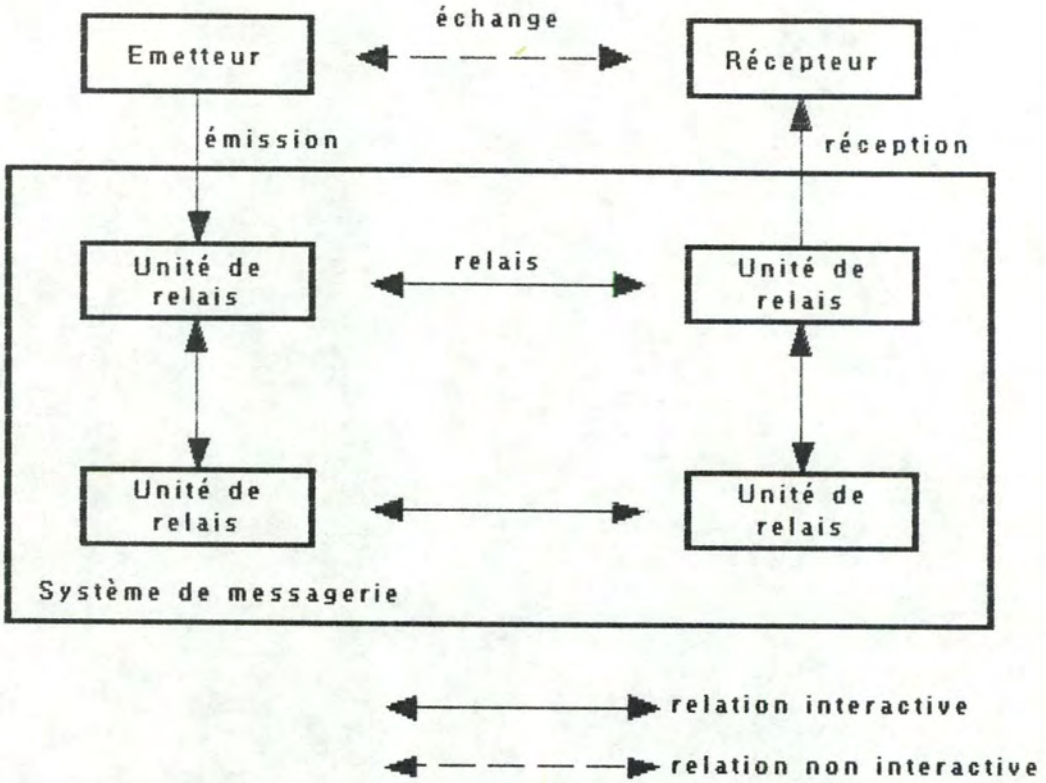


Figure 2.1

Système de messagerie

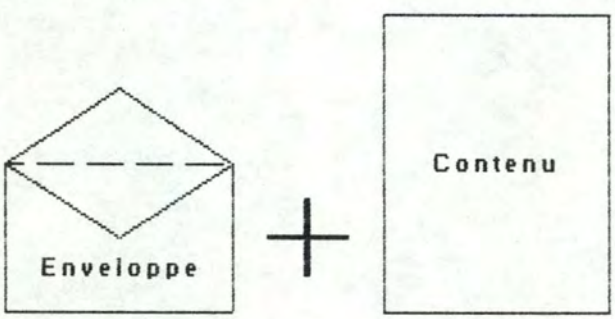


Figure 2.2

Message

Un message est constitué d'une enveloppe et d'un contenu (figure 2.2). L'enveloppe contient des indications utilisées pour la désignation du récepteur et l'acheminement de l'information vers celui-ci. Le contenu du message est l'information "utile" que l'émetteur désire faire parvenir au récepteur.

On peut mettre en évidence deux caractéristiques des systèmes de messagerie électronique :

- non-interactivité de l'échange

Un échange de messages n'est pas interactif, en ce sens qu'il n'existe pas de communication directe entre les utilisateurs. Il n'est donc pas nécessaire que le récepteur soit disponible ou connecté au système lors de l'émission d'un message qui lui est adressé. Le système prend en charge l'information, et l'achemine vers le récepteur. Celui-ci peut alors, lorsqu'il est à nouveau disponible, retirer le message.

Les utilisateurs ne doivent être connectés au système que pour les opérations d'émission et de réception de messages. Le transport de l'information ne nécessite pas d'interaction avec les utilisateurs (figure 2.1).

- acheminement des messages par relais (store and forward)

Certains systèmes de messagerie sont constitués d'un ensemble de noeuds de relais. L'acheminement d'un message d'un utilisateur vers un autre peut donc impliquer le passage par un certain nombre de ces noeuds (figure 2.1).

Chaque noeud de relais enregistre temporairement le message, pour ensuite le transmettre vers le noeud suivant, ou vers le récepteur. Ce mode de transport de l'information est généralement appelé store and forward (enregistrement et transmission).

Cette façon de procéder possède certains avantages. Tout d'abord, un utilisateur interagit toujours avec le même noeud de relais. Etant donné que ces noeuds peuvent être dispersés géographiquement, la distance entre l'utilisateur et le noeud de relais auquel il peut se connecter sera minimale. On peut ainsi réduire fortement les coûts de communication puisque les utilisateurs ne sont pas obligés d'interagir avec un noeud central très éloigné. D'autre part, le système est plus disponible et plus fiable, étant donné que la route que suit le message peut être choisie dynamiquement, pour éviter des noeuds de relais saturés ou tombés en panne.

Dans le cas d'un système de messagerie limité aux utilisateurs d'un seul ordinateur, le système n'est constitué que d'un seul noeud central, auquel accèdent tous les utilisateurs.

Bon nombre de constructeurs ont déjà proposé des systèmes de messagerie. Ils sont bien souvent incompatibles entre eux, et ceci pour diverses raisons.

Tout d'abord, les ordinateurs des différents constructeurs ne peuvent pas être interconnectés sans problèmes. De plus, les approches adoptées lors de l'implémentation des logiciels de messagerie ne sont pas nécessairement similaires, ce qui rend une collaboration difficile. C'est le cas par exemple pour les conventions d'adressage et de désignation des utilisateurs que l'on désire joindre, pour le format et le contenu des messages, ... Chaque système est fortement dépendant des choix qui ont été faits lors de sa conception, et ceci ne simplifie pas vraiment le problème de l'interconnexion. Ces problèmes sont mis en évidence dans [19].

C'est dans ce cadre que s'inscrit la série de recommandations X400 à X430 proposées par le CCITT sur la messagerie électronique.

On y trouve un modèle général de messagerie et une description des services qu'un tel système peut offrir [21] (recommandation X400). Dans [22] (recommandation X401), une description plus précise de ces services est proposée. La recommandation X408 [23] énonce un ensemble de règles de conversion du contenu de messages, intervenant lorsque ceux-ci sont échangés entre des utilisateurs disposant de périphériques télématiques différents. La recommandation X409 [24] propose un langage de description d'unités de donnée de protocole et une représentation concrète pour les valeurs de celles-ci. Ce langage est utilisé dans les autres recommandations de la série pour définir les unités des protocoles qu'on y propose. Nous présenterons au chapitre 3 le contenu de cette recommandation. La recommandation X410 [25] propose une description de deux entités fonctionnelles d'un système de messagerie. Dans [26] (recommandation X411), on trouve une description du protocole utilisé pour le relais des messages. La recommandation X420 [27] définit un protocole spécifiant les modalités d'échange de messages entre les utilisateurs. Enfin, la recommandation X430 [28] décrit les conventions d'accès auxquelles doivent se soumettre les utilisateurs disposant uniquement de terminaux télématiques.

Ces recommandations proposent un modèle de messagerie général et abstrait permettant d'intégrer la grande majorité des systèmes existants avec un minimum de modifications. Elles décrivent cependant de manière suffisamment précise les services à fournir à l'utilisateur, en vue d'assurer la compatibilité entre les systèmes qui respectent ces propositions.

La portée internationale de ces recommandations doit alors garantir que les implémentations de systèmes de messagerie qui les respectent puissent s'interconnecter sans problème.

La recommandation X400 [21] propose un modèle permettant de représenter les entités fonctionnelles d'un système de messagerie. Elle met en évidence comment ce modèle peut être mis en correspondance avec des architectures physiques et organisationnelles. Dans ce qui suit, nous présenterons ces trois approches. Nous désignerons un système de messagerie exprimé dans ce modèle, par système de messagerie X400.

Nous montrerons également comment ces entités fonctionnelles s'intègrent dans le modèle de référence OSI, abordé au chapitre 1.

2.2.1. Modèle fonctionnel

Le modèle fonctionnel [21] est destiné à mettre en évidence les composants d'un système de messagerie, en distingant leurs fonctionnalités et en détaillant les interactions possibles entre ces éléments.

Cette représentation est indépendante des caractéristiques d'implémentation du système. On ne tient en effet pas à préciser quel matériel (ordinateurs, terminaux, réseaux, ...) est mis en oeuvre ou quel composant logiciel est utilisé pour réaliser une certaine fonction. Les constructeurs sont ainsi libres d'implémenter le système comme ils l'entendent, pour autant que cette mise en oeuvre respecte la définition fonctionnelle proposée.

Le modèle fonctionnel est représenté schématiquement à la figure 2.3.

On retrouve dans ce modèle les utilisateurs, qui appartiennent à l'environnement de messagerie. A chaque utilisateur est associé une entité appelée Agent Utilisateur (User Agent ou UA). Cette entité est le représentant de l'utilisateur au sein du Système de Messagerie (Message Handling System ou MHS). Il s'agit d'un programme d'application qui aide l'utilisateur lors de la préparation des messages, et qui interagit avec le Système de Transfert de Messages (Message Transfer System ou MTS) pour demander l'acheminement de l'information que l'utilisateur émet, ou prendre en charge les messages qui lui sont destinés. L'information confiée au MTS peut transiter par un ou plusieurs noeuds de relais, appelés Agents de Transfert de Messages (Message Transfer Agent ou MTA).

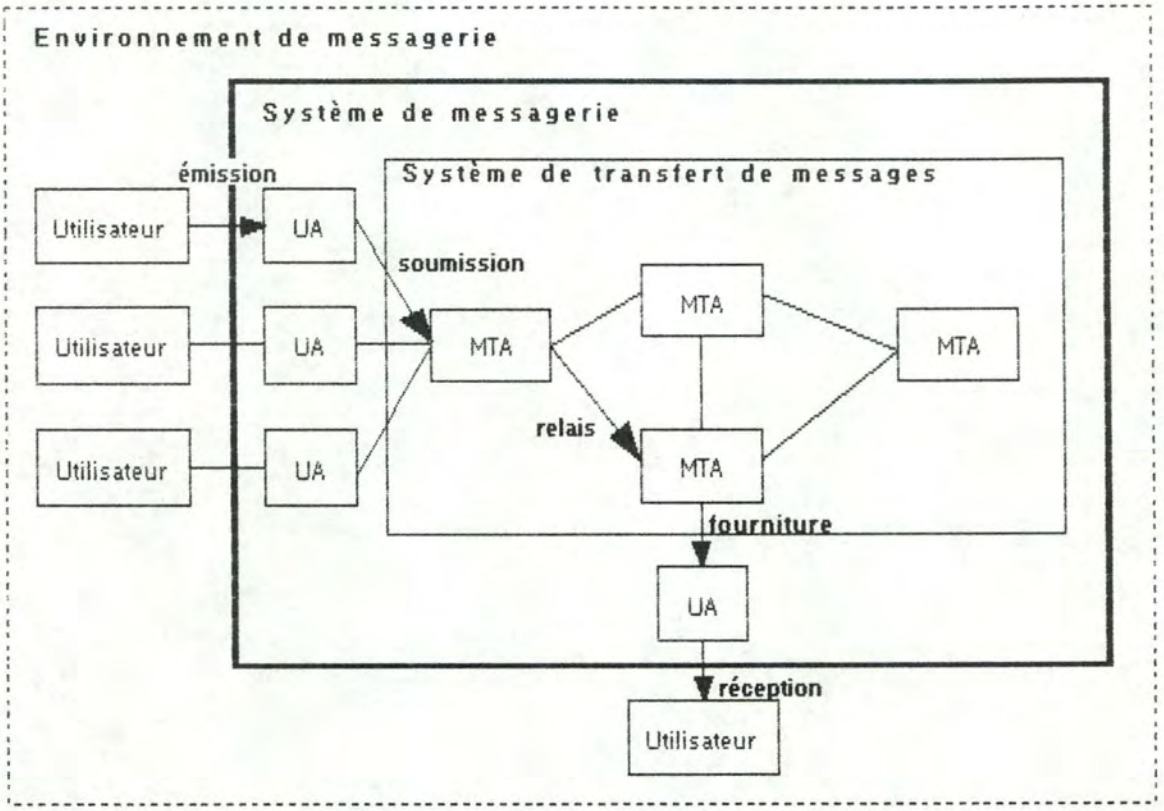


Figure 2.3

Modèle fonctionnel
d'un système de messagerie

Lorsqu'un émetteur désire faire parvenir de l'information à un utilisateur, il demande à son UA, par le biais d'une opération d'émission, d'envoyer un message. L'UA confie alors le message au MTS par l'intermédiaire d'une primitive de soumission. Pour ce faire, l'UA s'adresse au noeud MTA auquel il est relié. Le message est ensuite transporté d'un noeud de relais à l'autre, pour finalement parvenir au MTA auquel est connecté l'UA du récepteur.

Ces MTAs interagissent par le biais d'opérations de relais. Le dernier MTA sur la route suivie pour le transport du message, confie le message à l'UA du récepteur par l'intermédiaire d'une opération de fourniture. Le récepteur peut alors disposer du message envoyé par l'émetteur. Pour ce faire, il interagit avec son UA par le biais d'une opération de réception.

L'UA assume également des fonctions d'édition, d'affichage et de stockage des messages. Celles-ci sont appelées fonctions locales, et ne sont pas standardisées.

Les UAs de l'émetteur et du récepteur sont qualifiés d'UAs coopérants, étant donné que certains services offerts aux utilisateurs nécessitent la coopération de deux agents utilisateur. Il est clair que pour fournir correctement ces services, les UAs doivent respecter certaines conventions. Celles-ci sont décrites au moyen d'un protocole.

Dans la recommandation X400, un tel protocole est proposé dans le cadre de la messagerie interpersonnelle (InterPersonal Message handling System ou IPMS).

Les noeuds MTA du système de transfert de messages coopèrent en vue de faire parvenir un message soumis par l'UA de l'émetteur, à l'UA du récepteur.

La route du message est déterminée de manière incrémentale. Elle n'est pas fixée lors de la soumission. Ainsi, un noeud de relais détermine le MTA suivant sur la route, en respectant des critères tels que la charge du réseau, la distance à parcourir, ... De relais en relais, le message parvient à l'UA du récepteur.

Il est important de noter que l'enveloppe du message est susceptible de changer dans le temps. En effet, on distingue trois types d'enveloppe.

L'enveloppe de soumission contient des informations concernant l'UA de l'utilisateur à joindre, les types de services demandés, ...

L'enveloppe de relais comprend ces mêmes informations, accompagnées d'indications relatives au routage. Ainsi, chaque noeud indique par exemple son nom pour signaler aux MTAs suivants sur la route, que le message est déjà passé par ce noeud. Lorsqu'un MTA reçoit un message, il vérifie avant de le relayer que celui-ci n'est pas déjà passé par lui.

De plus, lorsqu'un même message est destiné à plusieurs utilisateurs, l'UA ne soumet qu'un seul message au MTS. Il spécifie cependant dans l'enveloppe les noms des UAs de tous les utilisateurs à joindre. Le message est transporté tout d'abord sur un chemin commun. Lorsque le message doit suivre une route différente pour un certain nombre d'utilisateurs récepteurs, le message est simplement dupliqué. Les nouveaux messages contiennent alors dans leur enveloppe de relais les références des UAs qu'ils doivent joindre sur cette nouvelle route. Les différentes copies du message sont ensuite acheminés via des routes différentes, pour finalement être fournies aux UAs des récepteurs.

L'enveloppe de fourniture contient d'autres informations, utilisées par l'UA du récepteur. Il s'agit par exemple d'indications sur la date et l'heure de soumission, sur les autres récepteurs du message, ...

L'évolution de l'enveloppe est retracée sur un exemple à la figure 2.4.

L'utilisateur U1 désire envoyer de l'information aux utilisateurs U2 et U3. Par le biais d'une opération d'émission, il signale sa demande à son agent utilisateur (UA1). Celui-ci détermine les noms des UAs (UA2 et UA3) des utilisateurs à joindre. UA1 indique ces références dans l'enveloppe et y joint l'information "utile" reçue de l'utilisateur, pour constituer le message M1. Celui-ci est soumis à MTA1, le noeud de relais auquel est relié UA1. MTA1 détermine que le noeud suivant est MTA2. Il lui envoie le message, après y avoir indiqué sa référence. Ainsi, si le message, par une quelconque circonstance, passait à nouveau par MTA1, celui-ci pourrait déterminer qu'une erreur s'est produite lors du routage.

MTA2 étudie alors l'enveloppe reçue, et détermine que le message doit être dupliqué. En effet, pour joindre UA2, il est nécessaire de relayer le message vers MTA3, tandis que pour le faire parvenir à UA3, le message doit passer par le noeud MTA4. Ainsi, le message M1 est dupliqué, pour donner M2 et M3. M2 contient dans l'enveloppe la référence de UA2, et M3 le nom de UA3. MTA2 indique dans l'enveloppe de M2 les noeuds de relais par lesquels était passé le message M1, et y ajoute sa référence. Il procède de la même manière pour l'enveloppe de M3. Ces deux nouveaux messages sont alors relayés respectivement vers MTA3 et MTA4. Le message M1 n'est plus utilisé.

Le noeud de relais MTA3, recevant le message M2, peut déterminer que l'UA auquel il est destiné, est un des UAs qu'il dessert. Il fournit donc le message à UA2, après avoir retiré de l'enveloppe les indications qui ont été nécessaires pour le routage. L'utilisateur U2 peut ensuite obtenir le contenu du message par le biais d'une opération de réception. L'évolution du message M3 est similaire.

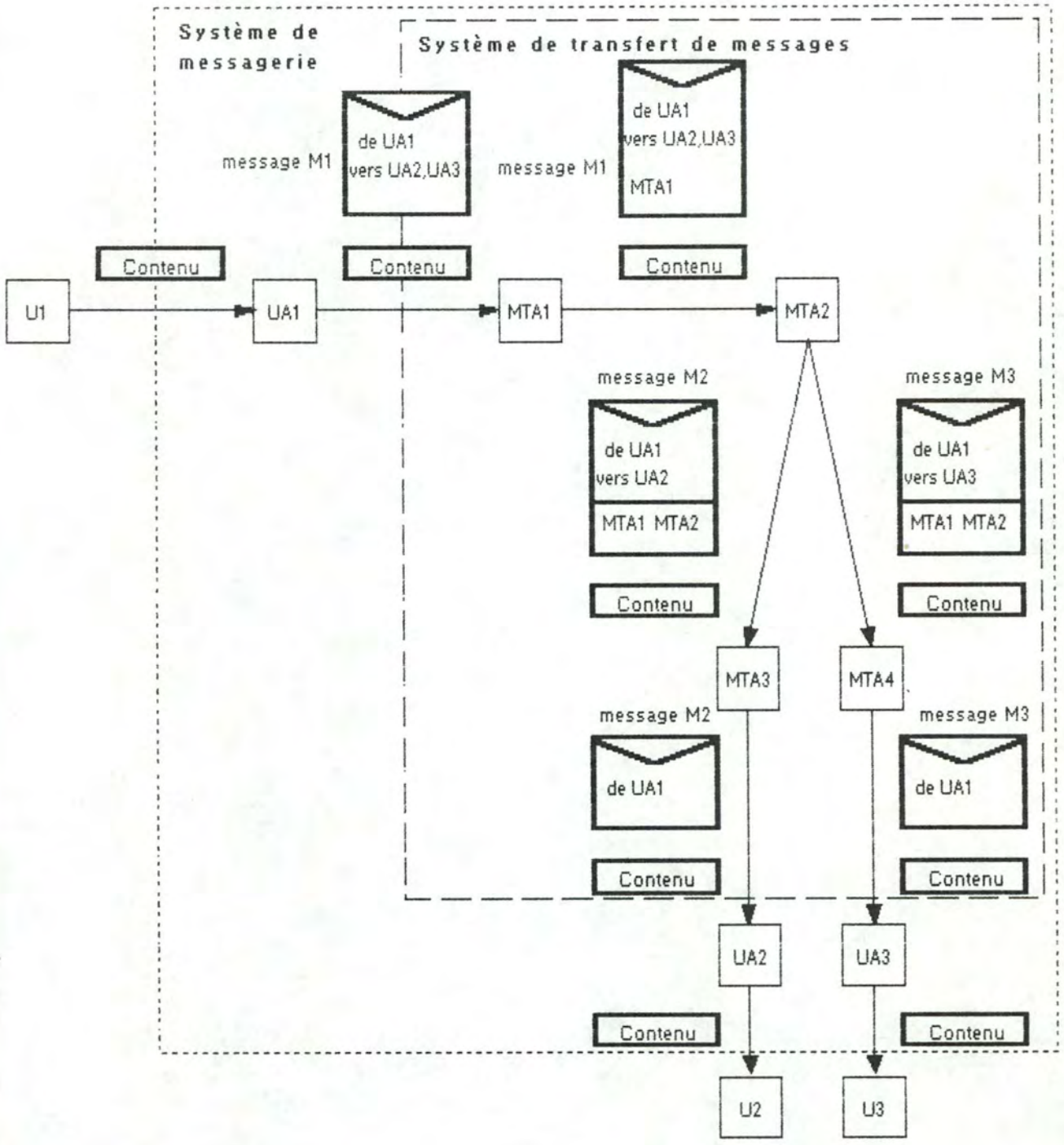


Figure 2.4

Evolution de l'enveloppe

2.2.2. Architectures physiques

Le modèle fonctionnel abordé au point 2.2.1 est une description générale et abstraite des composants existant au

sein d'un système de messagerie. Il ne tient pas compte de détails d'implémentation ou d'organisation physique.

La recommandation X400 [21] met cependant en évidence que ce modèle est une expression générale d'un grand nombre d'architectures physiques.

Nous présenterons dans ce qui suit quelques architectures physiques proposées dans cette recommandation, en mettant en évidence l'agencement des composants fonctionnels du modèle abordé au point 2.2.1.

L'utilisateur interagit avec son UA par l'intermédiaire de périphériques d'entrée/sortie (claviers, écrans, imprimantes, équipements fac-similé, ...).

L'UA est un programme d'application s'exécutant dans un système de traitement ou dans un terminal intelligent (figure 2.6).

Lorsque UAs et MTAs sont implémentés dans un même système de traitement, on les qualifie de co-résidents. Ici, l'agent utilisateur accède au service de transfert de messages par le biais d'un MTA se trouvant dans le même système de traitement (figure 2.5.)

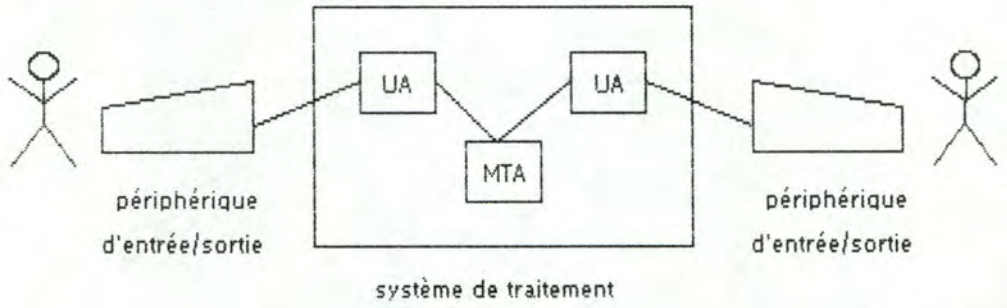


Figure 2.5

UAs et MTAs co-résidents

Un UA peut également être implémenté dans un système différent du système contenant le MTA. On qualifie alors l'UA d'isolé (figure 2.6). Dans ce cas, l'UA interagit avec le MTA via un réseau, en utilisant des protocoles spécifiques.

Il est également possible qu'un MTA soit seul dans un système de traitement (figure 2.6). Ce sera par exemple le cas pour un noeud de relais public.

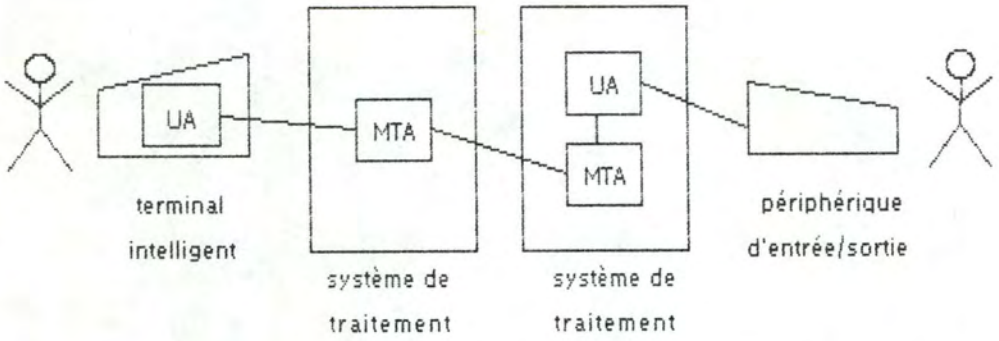


Figure 2.6

UAs et MTAs co-résidents et isolés

Une combinaison de ces architectures est représentée schématiquement à la figure 2.7. On y trouve un UA s'exécutant dans un terminal intelligent, des UAs implémentés dans un même système de traitement, et un UA et un MTA co-résidents. Les connexions entre ces systèmes de traitement se font par le biais de réseaux. Ceux-ci peuvent être de n'importe quel type.

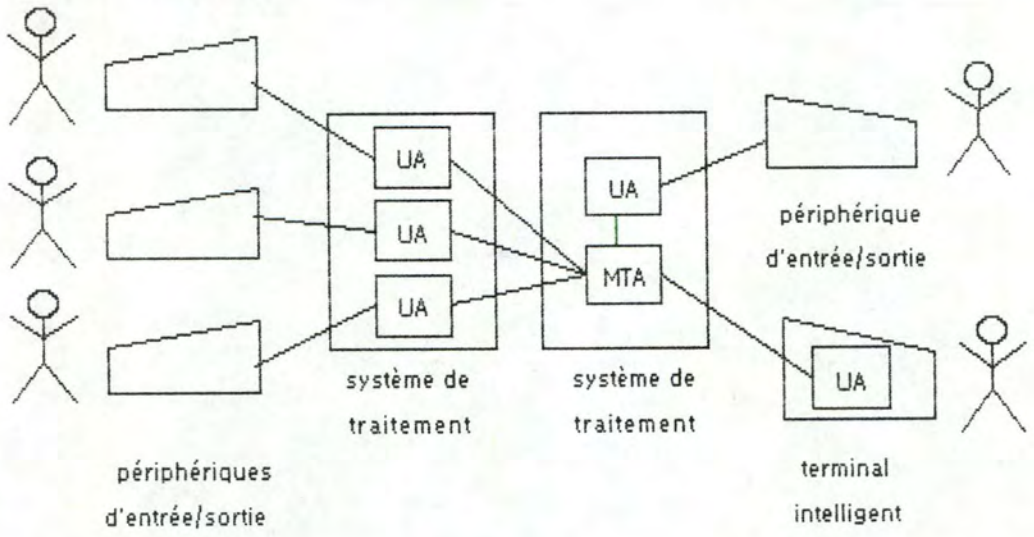


Figure 2.7

Combinaison des architectures physiques

2.2.3. Architectures organisationnelles

La recommandation X400 [21] définit également les rôles que peuvent jouer les administrations publiques et les organisations privées dans le cadre de la messagerie. On y présente différentes architectures organisationnelles pouvant découler du modèle fonctionnel présenté au point 2.2.1.

Ceci met en évidence comment il est possible, dans le cadre de ce modèle, d'intégrer des systèmes de messagerie existant déjà dans des administrations publiques ou des organisations privées.

On définit un **Domaine de Gestion** (Management Domain ou MD) comme étant une collection d'au moins un MTA et de zéro ou plusieurs UAs. On distingue les domaines de gestion publics (Administration Management Domain ou ADMD) et privés (Private Management Domain ou PRMD).

Un ADMD comprend les UAs et MTAs appartenant à une administration publique, tandis qu'un PRMD est constitué des UAs et MTAs d'une organisation privée.

Certaines contraintes sont imposées par la recommandation X400. Un domaine de gestion doit toujours être contenu complètement dans un seul pays. La communication entre MDs s'effectue par le biais des noeuds de relais MTA. Lorsqu'un message est acheminé d'un domaine de gestion public vers un autre, la responsabilité pour ce message est passée du premier au second MD. Cependant, lorsqu'un message transite d'un ADMD vers un PRMD, le domaine de gestion public reste entièrement responsable du relais au sein du domaine privé. De plus, un PRMD ne peut jamais relayer un message entre deux ADMDs.

Une architecture organisationnelle possible est représentée à la figure 2.8. On y distingue deux pays (A et B), des domaines de gestion privés (PRMD1,PRMD2 et PRMD3) et publics (ADMD1, ADMD2 et ADMD3). Toutes les contraintes énoncées plus haut sont respectées dans cet exemple.

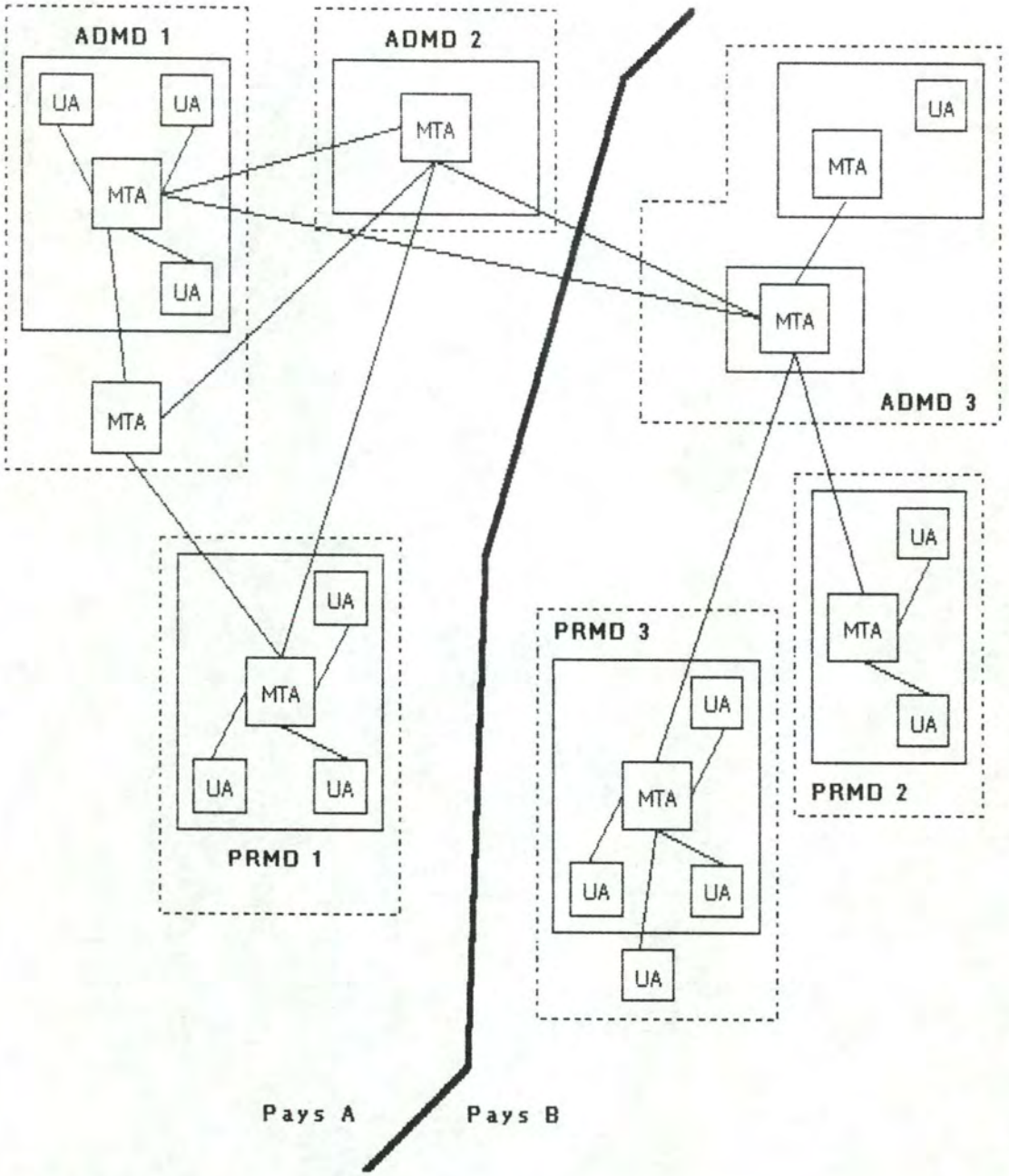


Figure 2.8

Exemple d'architecture organisationnelle

2.2.4. Messagerie X400 dans le modèle OSI

La messagerie X400, en vue d'assurer la compatibilité entre les différents systèmes qu'elle peut inclure, utilise l'architecture OSI présentée au chapitre 1.

Les applications de messagerie font partie de la couche sept du modèle OSI, et peuvent donc utiliser les services des couches inférieures pour entrer en communication avec d'autres applications de messagerie. Ceci est représenté schématiquement à la figure 2.9.

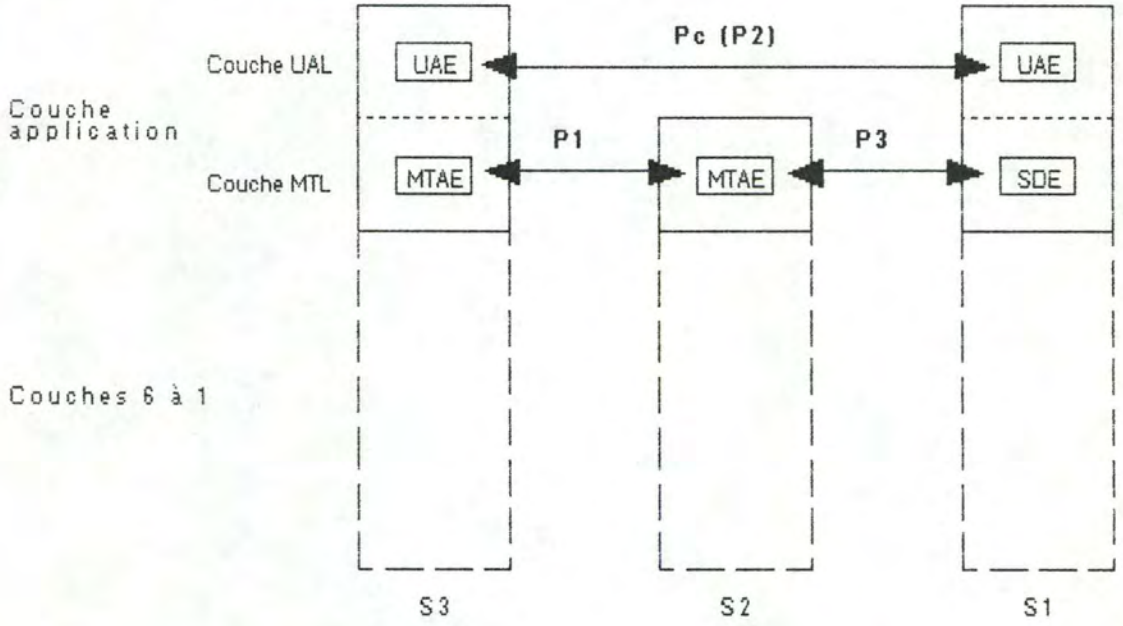


Figure 2.9

Applications de messagerie dans OSI

La couche application est elle-même subdivisée en deux couches : la couche de transfert de messages (Message Transfer Layer ou MTL) et la couche des agents utilisateur (User Agent Layer ou UAL).

La couche MTL offre un service de transfert de messages à la couche UAL. Cette dernière fournit un service d'émission/réception de messages aux utilisateurs de la messagerie.

Trois entités sont représentées à la figure 2.9 : L'entité UA (User Agent Entity ou UAE), l'entité MTA (Message Transfer Entity ou MTAE) et une entité de soumission/fourniture de messages (Submission/Delivery Entity ou SDE).

L'UAE réalise les fonctions de l'agent utilisateur, évoquées au point 2.2.1. Pour offrir le service

d'émission/réception de messages, elle dialogue avec l'UAE paire suivant un protocole spécifique aux types de contenus qu'elles peuvent s'échanger (Protocol for Contents ou Pc). Celui-ci définit les caractéristiques de l'information qu'un émetteur peut envoyer à un utilisateur. Ce protocole est par exemple P2, définissant les types d'informations que peuvent s'échanger des utilisateurs d'un système de messagerie inter-personnelle (IPMS). A l'heure actuelle, P2 est le seul protocole Pc qui soit défini.

Il faut noter que les protocoles Pc sont des protocoles "sans connexion", étant donné que les entités paires ne sont pas en liaison directe.

Pour offrir ce service d'émission/réception de messages, les entités UAEs paires utilisent également les services de la couche MTL.

La couche MTL peut contenir deux types d'entités : une entité réalisant les fonctions d'un MTA (MTAE), ou une entité de soumission/fourniture de messages (SDE). Cette dernière est nécessaire dans des systèmes ne disposant pas d'un MTA (S3). Cette entité permet à l'UAE d'accéder aux services du noeud de relais le plus proche pour la soumission et la fourniture de messages. Elle se charge de faire parvenir au MTAE les demandes de service émanant de l'UAE et vice-versa. Elle simule donc la présence d'un MTAE dans le même système de traitement, et l'UAE peut se comporter "comme si" le MTAE était directement disponible.

Les entités SDE et MTAE dialoguent suivant un protocole P3. Celui-ci décrit les conventions que ces deux entités doivent respecter pour permettre de fournir le service de transfert de message à l'UAE.

Les entités MTAEs communiquent suivant un protocole P1. Celui-ci définit exactement les modalités de relais de messages, en vue de fournir aux entités de la couche UAL le service de transfert de messages.

2.3. Services offerts par un système de messagerie

Les services offerts par un système de messagerie à ses utilisateurs sont nombreux et variés.

Un certain nombre de ceux-ci ne peuvent être offerts correctement que s'ils sont disponibles sur les différents systèmes de messagerie interconnectés au sein du MHS.

La recommandation X400 [21] définit un grand nombre de services que chacun des systèmes doit fournir. Ils peuvent ainsi, s'ils respectent ces propositions, valablement s'interconnecter.

Etant donné qu'un système de messagerie est constitué du système de transfert de messages (MTS) et du système de messagerie proprement dit (MHS), deux classes de services sont définies.

Les services de la première classe sont offerts par la couche MTL à la couche UAL. Celle-ci permet alors aux utilisateurs de disposer des services de la seconde classe. Les services offerts par la couche UAL ne sont cependant définis que dans le cadre d'un système de messagerie interpersonnelle (IMPS).

Nous évoquerons dans ce qui suit quelques services intéressants de chacune des deux classes. Le lecteur désirant de plus amples informations peut se référer à [21].

2.3.1. Services de transfert de messages

Le MTS permet aux UAs d'identifier les messages qu'ils soumettent ou qui leurs sont fournis. Ainsi, les UAs peuvent facilement faire référence à un message, en citant simplement l'identificateur que le MTS a attribué à celui-ci.

Cet identificateur est par exemple utilisé dans le cadre du service de "notification de non-fourniture". Le MTS indique ainsi à l'UA qu'un message précédemment soumis n'a pu être fourni à l'UA du récepteur. Le MTS fait référence à ce message par le biais de l'identificateur.

Le MTS peut également indiquer qu'un message soumis par l'UA a été fourni correctement à l'UA du récepteur. Il s'agit alors d'une "notification de fourniture".

Le système de transfert de messages peut fournir un service d'estampillage temporel. Ainsi, lorsqu'un UA soumet un message, le MTS y indique la date et l'heure de soumission. Celle-ci est alors disponible aux UAs de l'émetteur et du récepteur.

Comme nous l'avons déjà mis en évidence au point 2.2.1, un message soumis par l'UA de l'émetteur peut être destiné à plusieurs récepteurs. Un seul message est soumis, et le MTS

se charge de faire parvenir celui-ci aux différents UAs mentionnés dans l'enveloppe.

Le MTS permet à l'UA d'indiquer le caractère urgent du message qu'il soumet. Certains messages ne nécessitent pas d'acheminement rapide, tandis que d'autres doivent parvenir à l'UA du récepteur dans les plus brefs délais.

Lorsqu'un UA soumet un message, il peut spécifier que, si la référence de l'UA du récepteur qu'il a donnée n'est pas correcte, le message peut être fourni à un UA particulier dans le domaine de gestion du récepteur. Le choix de cet UA est effectué par ce domaine de gestion.

L'UA peut spécifier à la soumission d'un message que celui-ci ne peut en aucun cas être fourni à l'UA du récepteur avant une certaine date. Ce service est la "fourniture différée".

Lorsqu'un UA doit transmettre un message assez volumineux vers un UA dont il ne connait pas exactement les références ni les caractéristiques techniques, il peut demander au MTS de faire parvenir à cet UA un message sans contenu, possédant les mêmes caractéristiques que le message effectif. Ainsi, par le biais d'une notification de fourniture ou de non-fourniture, l'UA sait déterminer si le message est parvenu à l'UA du récepteur, et si les paramètres spécifiés étaient corrects. Il peut alors émettre le message effectif, étant donné qu'il est sûr qu'il parviendra à l'UA du récepteur.

2.3.2. Services de la messagerie interpersonnelle

Un certain nombre de services offerts aux utilisateurs du système de messagerie interpersonnelle IPMS sont similaires aux services présentés au point 2.3.1. L'utilisateur peut disposer d'un service d'identification de messages et de notification de réception/non-réception pour les messages qu'il émet. De plus, l'émetteur peut spécifier que si l'utilisateur auquel il désire faire parvenir un message n'existe pas, le message est retransmis automatiquement (auto-forward) vers un autre utilisateur.

D'autres services sont également disponibles. Ainsi, un utilisateur peut émettre un message vers un ou plusieurs récepteurs primaires (primary recipients) et faire parvenir des copies de celui-ci à des récepteurs de copies (copy recipients).

L'utilisateur peut indiquer au récepteur la durée de validité du message qu'il lui fait parvenir. Ainsi, après la

date spécifiée, le récepteur doit considérer que le message reçu précédemment n'a plus de signification.

Lorsqu'un utilisateur émet un message, il peut signaler à ses récepteurs le degré de confidentialité à accorder à ce message.

Un utilisateur a la possibilité de faire référence à d'autres messages par le biais de l'identificateur. Il indique par exemple au récepteur que l'information qu'il lui fait parvenir est une réponse à un message précédemment émis par cet utilisateur. Ainsi, le récepteur peut comparer le message qu'il avait émis et la réponse qu'il reçoit.

2.4. Messagerie électronique VidéoMail Service (VMS)

Un projet visant à mettre en place un système de messagerie Vidéomail Service (VMS) est développé conjointement par les Facultés Universitaires N.-D. de la Paix et la firme Electronique et Télécommunication Bell (ETB).

Ce système doit permettre à ses utilisateurs la composition, l'édition, l'archivage et l'échange de documents multimédia.

Un document multimédia est un ensemble structuré d'informations de types variés (texte, graphiques en deux et trois dimensions, voix, musique Hi-fi, images, ...). Ces différents composants peuvent avoir une relation spatio-temporelle entre eux.

Ce document peut être vu comme un message audio-visuel d'une qualité et d'une complexité comparables à celles d'un montage dia. On peut ainsi exprimer que lors de la visualisation du document, une telle image (graphique, diapositive, document télécopié, image vidéo, ...) soit affichée à l'écran, en même temps qu'un texte disposé à tel emplacement, et qu'un message sonore (voix, musique, ...) accompagne l'information graphique pendant un certain temps.

Le projet vise à concevoir une architecture de document permettant l'expression de ce genre de message. De plus, une station de travail multimédia doit être mise en place pour permettre la saisie, la consultation et l'édition d'un tel document. Elle est susceptible d'intégrer des composants matériels de tous types (caméras vidéo, microphones, chaîne Hi-fi, claviers, appareils dia, ...).

D'autre part, un document multimédia peut être acheminé vers un autre utilisateur par le biais d'un système de

messagerie X400. L'architecture de document doit permettre au récepteur de réutiliser certaines parties d'information contenues dans le document qu'il reçoit.

On peut envisager que la fonctionnalité d'échange de documents multimédia est réalisée par une entité, utilisant les services de la couche UAL dans un système de messagerie X400. On définit une classe de protocoles spécifiques à l'échange de documents multimédia, similaire à la classe de protocoles P_c exprimant les modalités d'échanges entre entités de la couche UAL.

3. Les langages formels de spécification

3.1. Introduction

Des efforts considérables sont entrepris par les organismes de standardisation pour tenter de normaliser au niveau international les protocoles de communication.

Cet objectif ne peut être atteint que par une description claire et précise du comportement que doivent respecter les systèmes communiquants. A l'heure actuelle, les textes officiels spécifiant ce comportement sont rédigés en langue naturelle, qui tolère des interprétations souvent fort diverses, sinon opposées.

L'ambiguïté de ces descriptions est due en grande partie au fait que deux personnes perçoivent généralement un texte en langue naturelle d'une manière différente. En vue de rendre les textes officiels moins ambigus, les organismes de standardisation envisagent d'utiliser des langages formels de spécification.

Les recommandations pourront à l'avenir être accompagnées d'une spécification utilisant un langage formel. Ceci permettra, en cas d'ambiguïtés, de se référer au texte formel, qui lui, ne permet normalement qu'une seule interprétation.

Ceci implique évidemment que le langage soit lui-même défini de manière claire et précise. De plus, la définition devra également être normalisée, de manière à éviter les interprétations contradictoires dûes à l'existence de versions distinctes du langage.

Un premier pas a été fait par l'introduction du langage X409 (voir point 3.3) qui permet la description des données échangées par un système avec son environnement. Les unités de données des protocoles de la messagerie électronique X400 ont été spécifiées par l'intermédiaire de ce langage. Il est probable que ce sera le cas pour d'autres protocoles de la couche application qui seront développés dans l'avenir.

Nous détaillerons dans ce qui suit les caractéristiques et qualités des langages formels, notamment face à la langue

naturelle. Ensuite, nous étudierons trois de ces langages : le langage X409 du CCITT et les langages LOTOS et ESTELLE de ISO.

3.2. Critères d'évaluation des langages formels de spécification

La description d'un système passe par deux étapes principales : décrire les données qu'il est susceptible d'échanger avec son environnement, et détailler le comportement que son environnement peut observer. Ces deux étapes peuvent être appliquées au système lui-même et à n'importe quel sous-système résultant d'une découpe plus fine du système global. On peut ainsi décrire un système avec différents niveaux de précision, suivant que l'on spécifie le système ou un de ses composants.

Les langages formels de spécification peuvent intervenir dans ces deux étapes de description, en ce sens qu'on peut utiliser leur formalisme pour exprimer la structure des données et le comportement observable. Certains langages permettent de décrire les deux aspects, d'autres ne sont applicables qu'à l'une des deux étapes.

Etant donné que les langages formels de spécification se distinguent par leurs bases théoriques, par la complexité de leur formalisme, par leur pouvoir d'expression, ... il est assez difficile d'évaluer a priori leurs qualités respectives.

On peut cependant définir un ensemble de critères qui permettent d'évaluer les qualités respectives des langages formels. Suite à la présentation des langages X409, LOTOS et ESTELLE, nous proposerons pour chacun d'eux une évaluation sur base de ces critères.

3.2.1. Applicabilité du langage à la spécification de systèmes communiquants

Le langage doit permettre la modélisation aisée de systèmes composés de modules indépendants. Ce critère est essentiel lorsqu'on considère l'organisation en couches du modèle OSI, où chaque couche est considérée par les autres comme une boîte noire indépendante qui n'interagit avec celles-ci que via des primitives de service.

Le langage doit permettre de faire abstraction des autres couches qu'on ne désire pas spécifier. Leur spécification se limitera à la description des primitives de service : pour chacune d'entre elles, on décrira leur effet et la manière dont il faut les invoquer. On supposera simplement qu'elles offrent le service correctement.

Il doit également être possible de décrire des couches dont les entités paires communiquent suivant un protocole "sans connexion" ou "orienté connexion", puisque ces deux modes de dialogue apparaissent dans le cadre du modèle OSI.

3.2.2. Facilité de modélisation et puissance d'expression

La modélisation d'un système dans un langage formel de spécification doit se faire de manière plus ou moins naturelle, en ce sens que le modèle doit découler presque directement de la perception informelle du système. Il ne faut pas que la spécification soit une construction artificielle n'ayant aucun lien direct avec le système réel.

Le langage doit offrir des possibilités d'expression suffisamment concise et riche. Concision et richesse du langage vont de pair car des expressions riches permettent d'exprimer plus avec un volume plus restreint de texte formel. La concision évitera de produire des spécifications "kilométriques" qui sont plus facilement sujettes à erreur. Mais la richesse des expressions ne doit pas être la source d'ambiguïtés ou d'interprétations erronées.

Enfin, ce langage doit pouvoir être appris et maîtrisé en un temps minimal. La complexité doit être raisonnable et en accord avec les possibilités d'expression.

3.2.3. Indépendance du langage par rapport aux décisions d'implémentation

Une qualité importante d'un langage formel de spécification est sa capacité de faire abstraction des détails d'implémentation. En effet, lorsqu'on spécifie un système, on désire décrire celui-ci en toute généralité, et non une de ses implémentations possibles. Les détails d'implémentation concernent souvent le choix d'une structure de données et l'organisation des traitements. Il est plus intéressant de pouvoir manipuler des données décrites de manière abstraite que de devoir choisir une structure bien particulière. Pour les traitements, il est également plus avantageux de définir leurs propriétés plutôt que leur enchaînement séquentiel.

Cette qualité se justifie surtout si l'on considère le principe de base du modèle OSI : garantir l'indépendance des constructeurs quant à l'implémentation, tout en assurant le respect des modalités de dialogue standard.

3.2.4. Communicabilité d'un langage formel de spécification

L'objectif majeur d'un langage formel de spécification est d'assurer la compréhension sans ambiguïté du système modélisé. Deux personnes lisant le même texte formel doivent interpréter celui-ci de la même manière et avoir une compréhension informelle similaire du système décrit [3].

Ainsi, si deux personnes implémentent ce système, les deux mises en oeuvre peuvent être différentes, mais elles assument les mêmes fonctionnalités.

Ceci est important dans le cadre de l'interconnexion des systèmes ouverts. En effet, deux constructeurs différents peuvent implémenter le modèle de référence en respectant les spécifications énoncées. Ainsi, bien que les mises en oeuvre soient différentes, les deux systèmes peuvent s'interconnecter sans problème. Ceci ne peut être atteint que si les spécifications sont claires et précises, de manière à être comprises de la même façon par les deux constructeurs.

La concision citée au point 3.2.2 possède le désavantage de la difficulté de compréhension directe du texte formel : plus les expressions sont riches, plus elles sont difficiles à comprendre. Un texte formel doit pouvoir être compris dans les grandes lignes à la première lecture. Il ne faut donc pas que le lecteur soit obligé de fournir un travail trop important pour arriver à un bon niveau de compréhension. La caractéristique des langages formels trop denses et trop complexes est le pourcentage élevé de commentaires en langue naturelle qui doit accompagner la spécification pour qu'elle soit compréhensible. Il faudrait que ce pourcentage reste modeste.

3.2.5. Disponibilité d'outils

Un langage formel de spécification peut devenir très utile lorsqu'il peut être combiné avec des outils permettant de vérifier que le texte formel respecte bien la définition du langage, et de générer automatiquement du code en langage évolué. On peut ajouter à cette liste un outil plus spécifique, permettant la production automatique de séquences de test. Ces séquences seront utilisées lors des tests de conformité de l'implémentation du protocole par rapport à la norme [4]. Cet outil a un domaine d'application plus restreint, puisqu'il n'intéresse a priori que les centres de test. La société qui produit un logiciel est évidemment plus intéressée, dans un premier temps, par la génération de code. Les séquences de test peuvent cependant être utiles après l'achèvement du logiciel pour permettre

aux constructeurs d'évaluer sa conformité par rapport à la norme qu'il est censé respecter [4].

Un langage pour lequel on ne dispose pas d'outils est néanmoins intéressant au point de vue de la communication sans ambiguïté. Un tel langage peut s'avérer utile dans le contexte des normes internationales : les organismes de standardisation pourraient joindre aux normes rédigées en langue naturelle une spécification formelle utilisant un langage plus ou moins répandu. Ceci permettrait d'éliminer un certain nombre d'ambiguïtés contenues dans le texte en langue naturelle, puisqu'on pourra toujours se référer au texte formel en cas de problèmes. Et ainsi, on n'impose pas au lecteur de ces normes de maîtriser le langage formel utilisé. Un langage formel de spécification permettra donc une communication plus rigoureuse des textes normalisés [4].

3.3. La recommandation X409

La recommandation X409 [24] fait partie des recommandations CCITT de la série X400 sur les systèmes de messagerie électronique. Un de ses objectifs est de définir un langage permettant la spécification abstraite de structures de données, et la désignation abstraite des valeurs de ces données.

Ce langage permet tout d'abord de définir de manière formelle les caractéristiques sémantiques des données qu'une entité peut échanger avec son environnement. Ces définitions de types de données sont abstraites, en ce sens qu'elles sont tout à fait indépendantes d'un certain codage ou d'une représentation physique déterminée.

Outre la définition de ces classes d'informations, le langage proposé dans la recommandation X409 permet de désigner un représentant d'une telle classe. Cette désignation est également indépendante de la représentation physique de la valeur.

Ces deux aspects exposés dans la recommandation X409 permettent qu'on utilise le langage dans la définition de syntaxes abstraites, qui permettent de désigner les informations échangées entre entités paires de la couche application du modèle OSI. Nous avons vu précédemment que ces entités doivent disposer d'un formalisme qui leur permette de spécifier des classes d'informations et de désigner des représentants de ces classes.

La désignation des types et des valeurs doit absolument être indépendante du format physique réel matérialisant ces informations. En effet, les entités paires sont libres de représenter les données qu'elles émettent comme elles

l'entendent. Il est donc nécessaire qu'elles disposent d'un moyen pour désigner des informations sans faire référence à un codage particulier.

Outre ce langage de définition de syntaxes abstraites, la recommandation X409 propose un format de codage permettant la représentation concrète des valeurs abstraites que l'on peut désigner dans ce langage. Ce format peut servir en tant que syntaxe concrète et il permettra de véhiculer entre les entités paires de la couche présentation les informations correspondant aux valeurs abstraites. Il faut noter que ce format n'est pas nécessairement celui qui est choisi pour l'enregistrement local des données par une entité application. C'est seulement lors de l'échange qu'intervient ce format standard.

Nous avons vu également que les entités paires de la couche présentation se chargent d'établir la correspondance entre les représentations physiques utilisées par les entités application lors de la communication. Cette fonctionnalité est assurée par la négociation d'une représentation commune (syntaxe concrète) intervenant dans le dialogue des entités présentation.

Le langage présenté dans X409, en plus de la définition abstraite des types de données et de la dénomination des valeurs, intègre la déclaration d'informations qui seront utilisées lors du codage physique des valeurs. Nous verrons plus loin qu'il s'agit de l'identification des types de données. Elle permet de distinguer dans le codage les types des valeurs et de les identifier par rapport à tous les autres types définis. Cette composante du langage n'est pas nécessaire au point de vue de la spécification abstraite des types de données, puisque leur dénomination seule suffit pour les distinguer. On peut dès lors considérer que le langage n'est vraiment abstrait que si on lui retire cette composante, étant donné que celle-ci établit un lien explicite avec une représentation particulière des données.

Nous allons maintenant présenter le langage proposé dans la recommandation X409 et aborder la représentation physique des valeurs. Il est préférable de présenter d'abord le format standard de représentation, puisque les exemples d'utilisation du langage X409 font référence à ce format, dans le but de visualiser les définitions abstraites de types et de valeurs de données.

L'exposé de la recommandation X409 est essentiellement un résumé du texte officiel [24]. Le lecteur désirant de plus amples informations peut se référer à [24] et [8].

3.3.1. La représentation standard : le codage TLV

Le format standard proposé dans la recommandation X409 est destiné à représenter les valeurs abstraites des types de données que l'on peut définir dans le langage X409. A chaque valeur correspond une représentation physique appelée élément de donnée, constitué d'une suite ordonnée de n octets, numérotés de 0 à n - 1. Chaque octet de l'élément de donnée peut être vu comme une suite ordonnée de bits, dont les bits de poids fort et de poids faible portent respectivement les numéros 8 et 1 (figure 3.1).

Pour permettre de coder des valeurs quelconques, il est nécessaire de pouvoir représenter des valeurs de longueurs variables. De plus, le codage doit être interprété conformément à la définition du type. C'est pourquoi le format standard utilise pour la représentation d'une valeur, trois composants constituant ensemble l'élément de donnée. Un premier composant indique le type de la valeur, un deuxième sa longueur et le dernier composant contient le codage de la valeur elle-même.

Figure 3.1

Principes de numérotation

Nous appellerons dans ce qui suit les composants respectivement composant T, composant L, composant V. Ils sont représentés schématiquement à la figure 3.2. On constate que le composant valeur peut être élémentaire ou constitué lui-même de plusieurs sous-éléments de donnée. Ce sont toujours les deux premiers composants qui permettent d'interpréter correctement l'ensemble des octets constituant le composant valeur.

La représentation standard X409 est parfois appelée codage TLV, rappelant ainsi la structuration des éléments de donnée en trois composants : Type, Longueur et Valeur.

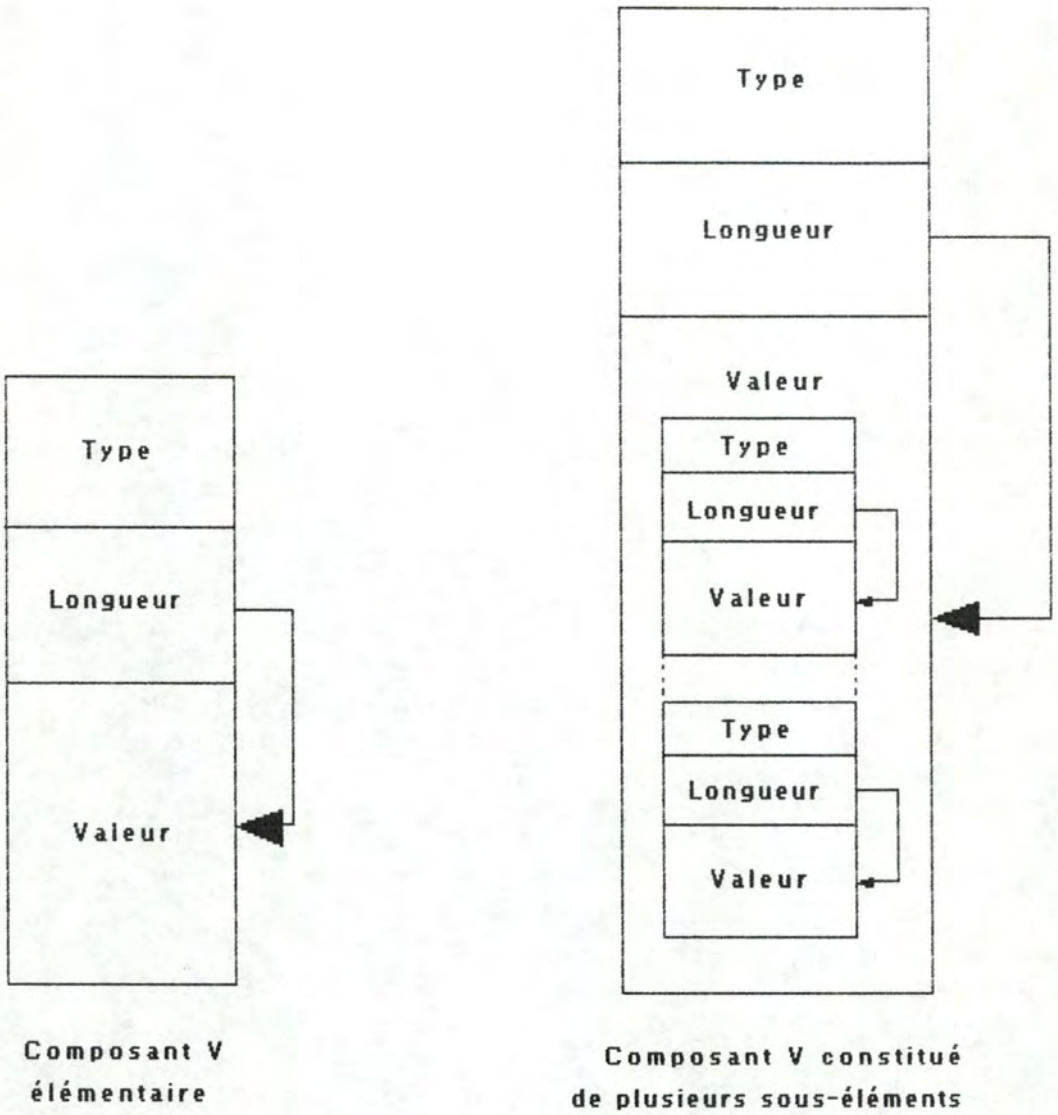


Figure 3.2

Eléments de données TLV

Nous allons maintenant décrire de manière précise ces trois composants.

3.3.1.1. Le composant TYPE

Le composant type doit permettre d'interpréter correctement la valeur codée. Pour ce faire, le type de la valeur doit pouvoir être distingué de tous les autres types existants. On utilise pour cela des identificateurs de type numériques. Pour éviter que l'ensemble des identificateurs devienne trop grand, il est subdivisé en sous-ensembles. Le type est dès lors identifié par la

classe et le numéro de l'identificateur. Ceci permet d'utiliser plusieurs fois le même nombre dans des classes différentes, sans pour autant diminuer la qualité de l'identificateur.

Chaque classe détermine la portée associée à l'identificateur. Elle est universelle, limitée à l'application que l'on spécifie, à usage privé ou limitée au contexte fourni par un autre type.

La classe universelle (UNIVERSAL) regroupe les identificateurs dont la portée est universelle. Ils sont utilisés dans toutes les applications, et ne peuvent dès lors être attribués que de manière standardisée.

La classe application (APPLICATION) contient les identificateurs qui ont une portée plus restreinte. Ils ne sont utilisés que dans le contexte de l'application à spécifier. Ces applications sont par exemple la messagerie X400, le transfert de fichiers,...

La classe privée (PRIVATE) est l'ensemble des identificateurs qui n'ont de signification que pour un nombre très restreint de personnes ou seulement dans le cadre d'une application utilisée au sein d'une organisation ou d'une entreprise. Ces identificateurs n'ont pas de signification en dehors de ce contexte privé.

La classe contexte (CONTEXT) est celle dont la portée est la plus limitée. Les identificateurs n'ont en fait de signification que dans un contexte très spécifique, qui est constitué d'un type dont le champ valeur est composé de plusieurs sous-éléments. Un identificateur de la classe contexte permet alors de distinguer les différents sous-éléments, dans le contexte défini par le type qui englobe ces sous-éléments.

Ces classes d'identificateurs sont distingués dans le codage du composant type via les bits 8 et 7 (bits cc) du premier octet. La correspondance entre classes et valeurs de ces bits est présentée à la figure 3.3. Le type d'une valeur est donc repéré par un identificateur d'une certaine classe dans un contexte déterminé.

Le composant type doit également fournir des informations concernant la forme du composant valeur. Il faut en effet savoir si la valeur est élémentaire, ou si elle est composée de plusieurs sous-éléments de donnée. Dans le premier cas, le type est dit primitif (PRIMITIVE), dans le second on le qualifie de constructeur

(CONSTRUCTOR). Cette information est codée dans le bit 6 (bit f) du premier octet du composant type (figure 3.4)

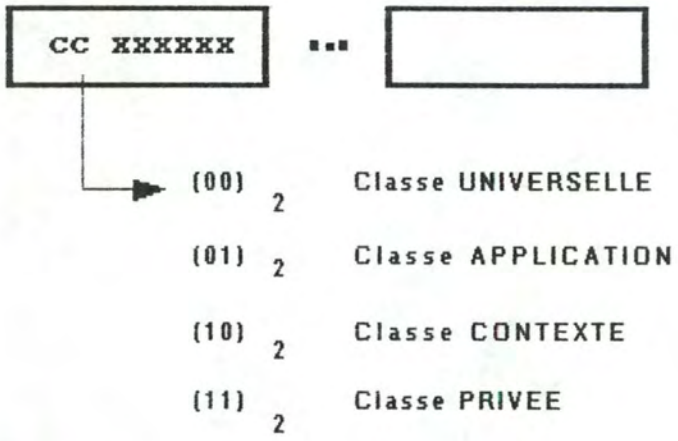


Figure 3.3

Classes d'identificateurs

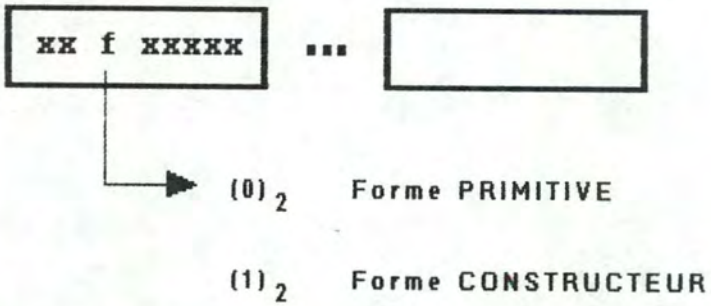


Figure 3.4

Formes de l'identificateur

Il reste maintenant à décrire comment la valeur de l'identificateur est codée dans le composant type. On utilise soit les 5 bits restant du premier octet (pour les identificateurs de 0 à 30), ou bien on étend le codage aux octets suivants dans le cas d'identificateurs plus grands. Cette extension est introduite par les bits 5 à 1 du premier octet qui valent $(11111)_2$. Chacun des octets suivants, sauf le dernier, ont le bit 8 égal à un. La valeur de l'identificateur correspond donc à la concaténation des bits 7 à 1 de ces octets. Ce codage est représenté schématiquement à la figure 3.5.

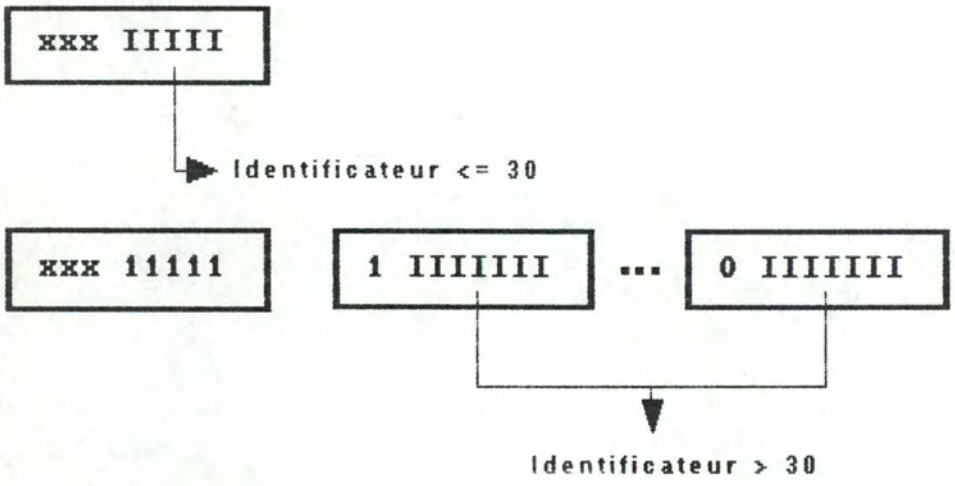


Figure 3.5

Identificateur numérique

Le codage des informations qui permettent la détermination du type de l'élément de donnée est résumé à la figure 3.6. C'est en connaissant exactement le type que l'on peut interpréter correctement le composant V de l'élément.

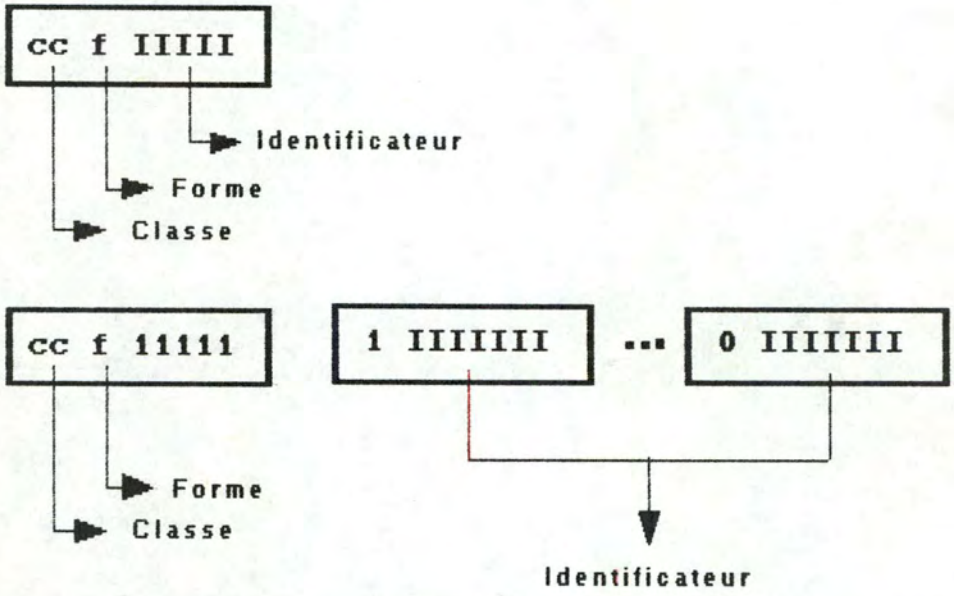


Figure 3.6

Codage du composant TYPE

3.3.1.2. Le composant LONGUEUR

Le composant longueur permet de déterminer la dimension exacte du composant valeur, exprimée en nombre d'octets. Trois formes de codage sont possibles : une forme courte, dans le cas où la longueur à exprimer n'excède pas 128 octets, une forme longue pour les longueurs dépassant ce seuil, et une forme indéfinie qui peut se substituer aux deux premières. Cette dernière forme n'est cependant permise que dans le cas d'un type constructeur. En effet, la fin du composant valeur ne peut être déterminée que par l'insertion, dans ce composant, d'un sous-élément spécial signalant la fin de la valeur.

Le codage de la forme courte s'effectue sur un seul octet, dans lequel le bit 8 vaut 0, et les bits 7 à 1 donnent la longueur. On obtient la configuration suivante :

$$(0\ LLLLLLL)_2$$

On peut ainsi coder la longueur 38 comme suit :

$$LG = 38 \qquad (0\ 0100110)_2$$

La forme longue est codée sur plusieurs octets. Elle est signalée par un bit 8 à 1 dans le premier octet. Les bits 7 à 1 du premier octet indiquent le nombre N d'octets nécessaires pour la représentation de la longueur. On obtient la configuration suivante :

$$(1\ NNNNNNN)_2 \quad (LLLLLLLL)_2 \quad \ldots \quad (LLLLLLLL)_2$$

On peut ainsi coder une longueur de 201 sous la forme suivante :

$$LG = 201 \quad N = 1 \qquad (1\ 0000001)_2\ (11001001)_2$$

La forme indéfinie est particulière en ce sens qu'on ne peut pas déterminer directement à partir du composant L la longueur du composant valeur. Cette forme est simplement annoncée par un octet de configuration binaire $(1\ 0000000)_2$ et un sous-élément spécial, nommé End Of Contents (EOC) indique la fin du composant valeur. Cette forme n'est cependant applicable que lorsque le composant valeur est constitué de plusieurs sous-éléments, l'indicateur EOC venant simplement s'ajouter comme dernier sous-élément. Ceci implique que le composant T indique, par l'intermédiaire du bit de forme (bit f), que le codage du composant V s'effectue sous forme constructeur.

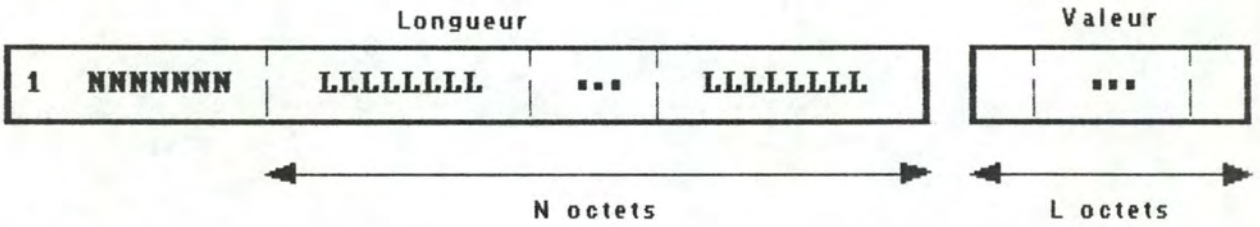
Etant donné que cet indicateur est lui-même un élément de donnée, il est également représenté par un codage TLV. La représentation est un élément de donnée de la classe universelle, d'identificateur zéro, de forme primitive, de longueur nulle, et dont le composant V est absent. La représentation codée est la suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	EOC	$(00)_{16}$	(00 0 00000)$_2$ — UNIVERSAL PRIMITIVE 0
L	nulle	$(00)_{16}$	(0 0000000)$_2$ — longueur sur un octet 0
V	absente		

Les différentes formes de longueurs et leur codages respectifs sont représentés schématiquement à la figure 3.7.

Forme courte

Forme longue



Forme indéfinie

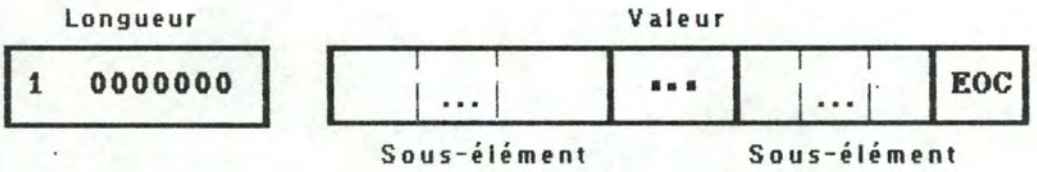


Figure 3.7

Codage du composant LONGUEUR

3.3.1.3. Le composant VALEUR

Le composant valeur est la substance même de l'élément de donnée représenté. Il se distingue des deux autres composants par le fait qu'il véhicule de l'information utile, à laquelle on peut accéder par les indications contenues dans les composants type et longueur.

La valeur codée est interprétée à l'aide des informations au sujet de son type. Elle peut être atomique, ou composée de plusieurs sous-éléments qui sont également codés suivant le format TLV.

3.3.2. Le langage X409

Le langage proposé dans la recommandation X409, que nous appellerons langage X409, permet de définir les types des données qu'une entité peut échanger avec son environnement. Dans le cadre du modèle OSI, l'entité appartient à la couche application, et son environnement est l'entité paire, qui est accessible via l'ensemble des couches intermédiaires.

Il faut noter qu'il existe un langage équivalent, proposé par l'organisme ISO. Il s'agit de l'Abstract Syntax and Notation One (ASN.1) [17] [18]. Ce langage intègre cependant quelques éléments supplémentaires que nous ne mettrons pas en évidence ici.

Le langage X409 permet de définir des types de données et de dénoter les valeurs de ces types. Pour que ceci soit possible, le langage intègre deux genres de notations standards : une notation de types et une notation de valeurs. La première permet de désigner des types pré-définis, de définir de nouveaux types et de les désigner. Ces nouvelles définitions se basent sur des types pré-définis, ou déjà définis par l'utilisateur. La notation de valeur détermine les règles syntaxiques de dénomination des valeurs des types pré-définis ou non.

Le langage donne également la possibilité de définir de nouvelles règles syntaxiques pour écrire des définitions de types et de valeurs. Ces règles de notation non standard sont introduites à l'aide d'une construction utilisant le concept de "macros" (point 3.3.2.3).

L'expression des règles d'écriture pour le langage X409 utilise un formalisme Backus Naur Form (BNF). Celui-ci est utilisé de manière générale pour décrire un langage formel. Une grammaire constituée d'un ensemble de règles syntaxiques permet de déterminer toutes les instances correctes du langage. Ces règles sont rédigées à l'aide de symboles qui peuvent être des symboles terminaux, qui apparaissent tels quels dans le langage, des symboles non-terminaux, équivalents à une suite de symboles (terminaux ou non), ou des opérateurs. Les règles syntaxiques sont appelées règles de production, et s'écrivent de la manière suivante :

$$A ::= B_1 \ldots B_n \mid C_1 \ldots C_m$$

Cette règle indique que le symbole non-terminal A peut être remplacé soit par la suite de symboles $B_1 \ldots B_n$, soit par la suite $C_1 \ldots C_m$. Les B_i et C_j peuvent être des terminaux, ou eux-mêmes des symboles non-terminaux, définis par d'autres règles de production. Les symboles ::= et | sont des opérateurs et ne font pas partie du langage défini. Le symbole ::= indique que le membre de gauche est

équivalent à une des suites de symboles évoquées dans le membre de droite, et séparées par l'opérateur |. L'application successive de l'ensemble des règles détermine une suite de symboles terminaux, qui représente une instance correcte du langage.

L'exemple suivant définit un langage permettant de dénoter des valeurs de types de données.

```
ValeurType       ::= ValeurBooléenne
ValeurBooléenne  ::= VRAI | FAUX
```

Cette grammaire définit les instances correctes du langage, qui sont respectivement VRAI et FAUX. Elles sont déduites par l'application successive des deux règles. L'expression "TRUE" n'est pas une instance correcte, étant donné qu'elle ne peut pas être déduite par l'application successive des règles de production.

Dans la recommandation X409, plusieurs symboles non-terminaux ont été définis une fois pour toutes, pour éviter de surcharger la grammaire complète du langage X409. Il s'agit de symboles fréquemment utilisés désignant des identificateurs (identifier), des entiers non signés (number), des chaînes de caractères (string), et des chaînes vides de symboles (empty) :

- identifier est un symbole qui représente toute suite d'au moins un caractère (lettres minuscules et majuscules, chiffres, tirets, ...) commençant par une lettre.

  Ex : MessageHandlingSystems-X400

- string est un symbole qui représente toute suite de zéro ou plusieurs caractères quelconques

  Ex : 01234ABC_D- ?

- number est un symbole qui représente toute suite d'au moins un chiffre en base dix, commençant par un chiffre non nul, sauf dans le cas ou le symbole désigne la valeur zéro.

  Ex : 123

- empty est un symbole qui désigne la suite vide de symboles. Il est souvent utilisé lorsqu'une règle syntaxique évoque plusieurs possibilités. Dans l'exemple suivant, on définit une liste d'éléments quelconques de

manière récursive. Le symbole empty est le cas trivial qui termine l'application récursive des règles.

Ex : Liste ::= Elément | Liste Elément
Elément ::= QuelqueChose | empty

Dans [18], trois autres symboles sont définis. Nous les utiliserons au même titre que les non-terminaux définis dans la recommandation X409 lorsque nous donnerons les règles syntaxiques du langage. Il s'agit des symboles cstring, hstring et bstring :

- cstring : (character string) est un symbole qui représente toute constante de type "chaîne de caractères". Celle-ci est constituée d'une suite de caractères quelconques, entourée de guillemets (").

Ex : "Transfert de fichiers"

- bstring : (binary string) est un symbole qui représente toute constante de type "chaîne de bits". Celle-ci est constituée d'une suite de zéros et de uns, entourée d'apostrophes (') et terminée par B, pour signaler que la constante est exprimée en binaire

Ex : '0100' B

- hstring : (hexadecimal string) est un symbole qui représente toute constante de type "chaîne de positions hexadécimales". Celle-ci est constituée d'une suite de caractères représentant un code hexadécimal (0 à 9 et A à F), entourée d'apostrophes (') et terminée par H, pour signaler que la constante est exprimée en hexadécimal

Ex : '0A9F' H

Dans la recommandation X409, on utilise d'autres formes pour désigner ces constantes. On fait apparaître les délimiteurs (apostrophes, guillemets , B et H) explicitement dans le texte de la grammaire.

"string" est donc équivalent à cstring lorsque string représente une suite de caractères. 'string' B est équivalent à bstring lorsque string représente une suite de positions binaires. 'string' H est équivalent à hstring lorsque string représente une suite de caractères hexadécimaux.

Nous utiliserons dans ce qui suit les non-terminaux cstring, bstring et hstring, plutôt que les constructions proposées dans la recommandation X409. Ces définitions nous semblent plus rigoureuses et plus précises.

Dans ce qui suit, nous considérerons tout d'abord les règles d'écriture des types et valeurs pré-définis. Nous exposerons ensuite les règles syntaxiques permettant la définition de nouveaux types et la dénomination des valeurs correspondantes.

La grammaire du langage X409 relative aux concepts abordés ici fait l'objet de l'annexe A.

3.3.2.1. Types de données pré-définis

Une série de types pré-définis sont disponibles dans le langage X409 pour la spécification de nouveaux types. Il s'agit des types BOOLEAN, INTEGER, BIT STRING, OCTET STRING, NULL, SEQUENCE, SET et CHOICE.

Nous donnerons pour chacun d'eux les règles de notation des types et des valeurs. Ainsi pour un type "xxx", elles seront présentées sous la forme suivante :

```
xxxType  ::= notation de type
xxxValue ::= notation de valeur
```

Nous aborderons d'abord les types pré-définis primitifs (BOOLEAN, INTEGER, BIT STRING, OCTET STRING et NULL) puis les types pré-définis constructeurs (SEQUENCE, SET et CHOICE), faisant intervenir plusieurs autres types.

Le type BOOLEAN

Le type BOOLEAN permet de modéliser les valeurs de vérité dénommées TRUE et FALSE. On en déduit assez aisément les notations standards suivantes :

```
BooleanType  ::= BOOLEAN
BooleanValue ::= TRUE | FALSE
```

La représentation standard d'une valeur de ce type est un élément de donnée de la classe universelle, de

forme primitive, d'identificateur 1, et dont le composant V est un seul octet. La valeur FALSE est représentée par 8 bits nuls, la valeur TRUE par toute autre configuration binaire.

On représente par exemple la valeur TRUE de type BOOLEAN de la manière suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	BOOLEAN	$(01)_{16}$	$(00\ 0\ 00001)_2$ UNIVERSAL PRIMITIVE 1
L	1	$(01)_{16}$	$(0\ 0000001)_2$ longueur sur un octet 1
V	TRUE	$(FF)_{16}$	$(1111\ 1111)_2$

Le type INTEGER

Le type INTEGER représente un entier. On peut également spécifier que l'entier ne peut prendre que quelques valeurs bien précises. Celles-ci sont énoncées en leur attribuant des noms de référence. Les règles de notation de type sont les suivantes :

```
IntegerType      ::= INTEGER |
                     INTEGER { NamedNumberList }
NamedNumberList  ::= NamedNumber |
                     NamedNumberList , NamedNumber
NamedNumber      ::= identifier (number) |
                     identifier (-number)
```

Les notations de valeur possibles sont donc soit un nombre (signé ou non), soit un des noms de référence qui ont été déclarés comme étant équivalents à des nombres. Ces notations respectent la grammaire suivante :

```
IntegerValue ::= number | - number | identifier
```

La représentation standard d'une valeur du type INTEGER est un élément de donnée de la classe universelle, de forme primitive et d'identificateur 2. Le composant valeur contient un entier représenté par un

nombre binaire complémenté à 2, dont le bit de poids fort est le bit 8 du premier octet, et celui de poids faible, le bit 1 du dernier octet.

On représente par exemple la valeur entière 256 de la manière suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	INTEGER	$(02)_{16}$	(00 0 00010)$_2$ UNIVERSAL PRIMITIVE 2
L	2	$(02)_{16}$	(0 0000010)$_2$ longueur sur un octet 2
V	256	$(01\ 00)_{16}$	$(0000\ 0001\ \ 0000\ 0000)_2$

On peut également, pour un type INTEGER {low(0),medium(1),high(2)}, désigner la valeur medium. Elle est équivalente à la valeur 1 et codée de la manière suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	INTEGER	$(02)_{16}$	(00 0 00010)$_2$ UNIVERSAL PRIMITIVE 2
L	1	$(01)_{16}$	(0 0000001)$_2$ longueur sur un octet 1
V	medium	$(01)_{16}$	$(0000\ 0001)_2$

Le type BIT STRING

Le type BIT STRING modélise un ensemble ordonné de zéro ou plusieurs valeurs de vérité, matérialisées par des bits. On peut également assigner des noms de

référence aux bits constituant l'ensemble. La notation du type respecte la grammaire suivante :

```
BitStringType    ::= BIT STRING |
                     BIT STRING { NamedNumberList }
NamedNumberList  ::= NamedNumber |
                     NamedNumberList,NamedNumber
NamedNumber      ::= identifier (number)
```

Les notations de valeur suivent ces deux possibilités de définition de type. On peut exprimer une valeur soit comme une suite de 0 ou de 1, encadrée d'apostrophes et suivie de la lettre B (bstring), ou comme une chaîne de caractères hexadécimaux entourée d'apostrophes et terminée par H (hstring), soit comme une liste de noms de référence. Si certains de ces noms ne sont pas mentionnés dans la désignation de valeur, les bits non cités sont supposés être nuls. On peut énoncer les règles de syntaxe suivantes :

```
BitStringValue ::= bstring | hstring |
                   { IdentifierList }
IdentifierList ::= identifier |
                   IdentifierList , identifier
```

La représentation standard d'une valeur du type BIT STRING est un élément de données de la classe universelle, de forme primitive et d'identificateur 3. Le composant valeur est une suite d'octets qui contient l'ensemble des bits. Etant donné que le nombre de bits n'est pas nécessairement un multiple de huit, un octet supplémentaire (le premier) mentionne le nombre de bits non utilisé dans le dernier octet.

On représente par exemple l'ensemble de 10 bits '1010110111' B comme suit :

	Interprétation	Codage hexadécimal	Codage binaire
T	BIT STRING	$(03)_{16}$	(00 0 00011)$_2$ UNIVERSAL PRIMITIVE 3
L	3	$(03)_{16}$	(0 0000011)$_2$ longueur sur un octet 3
V	'1010110111'B	$(06\ ADC0)_{16}$	(0000 0110 1010 1101 1100 0000)$_2$ 6 bits non utilisés dans le dernier octet

Pour un type BIT STRING {married(0),employed(1)}, la valeur {employed}, équivalente à '01' B, est représentée de la manière suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	BIT STRING	$(03)_{16}$	(00 0 00011)$_2$ UNIVERSAL PRIMITIVE 3
L	2	$(02)_{16}$	(0 0000010)$_2$ longueur sur un octet 2
V	{employed}	$(06\ 40)_{16}$	(0000 0110 0100 0000)$_2$ 6 bits non utilisés dans le dernier octet

Il faut noter que la recommandation X409 propose une alternative dans la représentation des valeurs du type BIT STRING. On peut en effet considérer que le type est de forme constructeur et que le composant valeur est constitué de plusieurs sous-éléments. La concaténation des composants valeur de ces sous-éléments fournit la valeur du BIT STRING. Cette forme de représentation peut se justifier dans le cas de valeurs très longues, où l'entité qui encode les données ne dispose pas directement de la longueur totale. Cette représentation utilise souvent une forme indéfinie de la longueur. On peut donc représenter la valeur '0A 3B 5F 29 1C D' H de deux manières :

Forme primitive

	Interprétation	Codage hexadécimal	Codage binaire
T	BIT STRING	$(03)_{16}$	$(00\ 0\ 00011)_2$ — UNIVERSAL PRIMITIVE 3
L	7	$(07)_{16}$	$(0\ 0000111)_2$ — longueur sur un octet 7
V	'0A3B5F291CD'H	$(04\ 0A3B5F\ 291CD0)_{16}$	$(0000\ 0100\ \ 0000\ 1010\ \ 0011\ 1011\ \ 0101\ 1111\ \ 0010\ 1001\ \ 0001\ 1100\ \ 1101\ 0000)_2$ — 4 bits non utilisés dans le dernier octet

Forme constructeur

		Interprétation	Codage hexadécimal	Codage binaire
T		BIT STRING (constructeur)	$(23)_{16}$	(00 1 00011)$_2$ — UNIVERSAL, CONSTR., 3
L		indéfinie	$(80)_{16}$	(1 0000000)$_2$
V	T	BIT STRING (primitif)	$(03)_{16}$	(00 0 00011)$_2$ — UNIVERSAL PRIMITIVE 3
	L	3	$(03)_{16}$	(0 0000011)$_2$ — longueur sur un octet, 3
	V	Valeur V1	$(00\ 0A3B)_{16}$	(0000 0000 0000 1010 0011 1011)$_2$ — pas de bit non utilisé
	T	BIT STRING (primitif)	$(03)_{16}$	(00 0 00011)$_2$ — UNIVERSAL PRIMITIVE 3
	L	5	$(05)_{16}$	(0 0000101)$_2$ — longueur sur un octet, 5
	V	Valeur V2	$(06\ 5F29\ 1CD0)_{16}$	(0000 0110 0101 1111 0010 1001 0001 1100 1101 0000)$_2$ — 4 bits non utilisés dans le dernier octet
	T	EOC	$(00)_{16}$	(00 0 00000)$_2$ — UNIVERSAL PRIMITIVE 0
	L	nulle	$(00)_{16}$	(0 0000000)$_2$ — longueur sur un octet, 0
	V	absente		

Dans la première forme, la valeur du BIT STRING est donnée par le composant V de l'élément de donnée. Dans la seconde, on obtient cette valeur par la concaténation des composants V des deux premiers sous-éléments de donnée (valeur V1 et valeur V2). Il est cependant nécessaire de supprimer le premier octet de V1 et de V2, et d'ajouter un octet mentionnant le nombre de bits non utilisés dans le dernier octet. Celui-ci est identique au premier octet de V2.

Le type OCTET STRING

Le type OCTET STRING modélise un ensemble ordonné de zéro ou plusieurs octets. Une valeur peut être désignée par une chaîne de caractères binaires encadrée d'apostrophes (') et terminée par B (bstring) ou par une chaîne de digits hexadécimaux entourée d'apostrophes (') et terminée par H (hstring). On peut également dénoter une valeur par une chaîne de caractères, entourée de guillemets ("), dont le code ASCII détermine la valeur de chacun des octets (cstring). Cette dernière possibilité n'est cependant conseillée que dans les cas où l'on est sûr que le code choisi par défaut correspond bien à la valeur que l'on veut représenter. Les notations standards des types et des valeurs respectent la grammaire suivante :

```
OctetStringType  ::= OCTET STRING
OctetStringValue ::= bstring | hstring |
                     cstring
```

La représentation standard d'une valeur du type OCTET STRING est un élément de donnée de la classe universelle, de forme primitive et d'identificateur 4. Le composant V est constitué d'une suite d'octets dont la valeur est celle spécifiée. Lorsque la notation de valeur est incomplète, les bits non spécifiés sont supposés nuls. Ceci permet d'assurer que la valeur peut être codée sur un nombre entier d'octets.

On représente par exemple la valeur '48 69 20' H de la manière suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	OCTET STRING	$(04)_{16}$	(00 0 00100)$_2$ — UNIVERSAL PRIMITIVE 4
L	3	$(03)_{16}$	(0 0000011)$_2$ — longueur sur un octet, 3
V	'48 69 20' H	$(48\ 69\ 20)_{16}$	(0100 1000 0110 1001 0010 0000)$_2$

La recommandation X409 propose, tout comme pour le type BIT STRING, une alternative dans la représentation des valeurs. On peut en effet coder une valeur sous forme constructeur, auquel cas la concaténation des composants V des sous-éléments de donnée détermine la valeur représentée.

Le type NULL

Le type NULL représente une donnée non existante. Les notations de type et de valeur sont confondues, et respectent la grammaire suivante :

```
NullType  ::= NULL
NullValue ::= NULL
```

Ce type particulier est généralement utilisé dans le type CHOICE, que nous aborderons plus loin.

La représentation standard de la valeur NULL est un élément de donnée de la classe universelle, de forme primitive et d'identificateur 5. Le composant valeur est inexistant, ce qui est signalé par une longueur nulle.

La représentation de la valeur NULL du type NULL est la suivante :

	Interprétation	Codage hexadécimal	Codage binaire
T	NULL	$(05)_{16}$	(00 0 00101)$_2$ — UNIVERSAL PRIMITIVE 5
L	nulle	$(00)_{16}$	(0 0000000)$_2$ — longueur sur un octet 0
V	absente		

Le type SEQUENCE

Le type SEQUENCE permet de modéliser un ensemble ordonné de zéro ou plusieurs valeurs, appelées éléments.

Plusieurs constructions sont possibles. On peut spécifier que les éléments sont tous du même type (SEQUENCE OF), ou bien qu'ils ont des types distincts (SEQUENCE { ElementTypes }). Dans ce cas, il faut explicitement spécifier les caractéristiques de ces éléments. Le type SEQUENCE où l'on ne spécifie rien (SEQUENCE) est équivalent à "SEQUENCE OF ANY". La grammaire de la notation des types est la suivante :

```
SequenceType ::=  SEQUENCE | SEQUENCE OF Type
                  SEQUENCE { ElementTypes }
```

Les éléments de la SEQUENCE sont soit présent à chaque fois, soit éventuellement omis de la notation de valeur lorsque l'on a spécifié qu'ils sont optionnels. Dans ce cas, on peut en plus définir que l'élément prend une valeur par défaut s'il est absent de la notation de valeur. La grammaire définissant les règles de spécification des éléments de la SEQUENCE est la suivante :

```
ElementTypes        ::= OptionalTypeList | empty

OptionalTypeList    ::= OptionalType |
                        OptionalTypeList , OptionalType

OptionalType        ::= NamedType |
                        NamedType OPTIONAL |
                        NamedType DEFAULT Value |
```

```
                                ComponentsOf
NamedType           ::= identifier Type | Type

ComponentsOf        ::= COMPONENTS OF Type
```

On constate qu'il est également possible d'utiliser les définitions d'éléments d'un autre type SEQUENCE, déjà défini par l'utilisateur, par l'intermédiaire de la construction COMPONENTS OF. On inclut simplement les déclarations relatives à ces éléments à l'endroit où est mentionné le mot-clé COMPONENTS OF.

La notation des valeurs d'un type SEQUENCE respecte la grammaire suivante :

```
SequenceValue       ::= { ElementValues }

ElementValues       ::= NamedValueList | empty

NamedValueList      ::= NamedValue |
                        NamedValueList , NamedValue

NamedValue          ::= identifier Value | Value
```

La spécification d'un type SEQUENCE {INTEGER,INTEGER,BOOLEAN} définit que celui-ci est un ensemble constitué de trois éléments. Le premier et le second éléments sont de type INTEGER et le troisième est de type BOOLEAN. Une valeur d'un tel type est par exemple {123,321,TRUE}, ce qui signifie que la valeur du premier élément est 123, que le deuxième élément vaut 321 et que le dernier prend la valeur TRUE.

Il est possible de déterminer sans ambiguïté la valeur de chacun des éléments, étant donné que l'ordre utilisé pour énoncer les valeurs de chacun d'eux doit être le même que l'ordre dans lequel les éléments sont définis dans le type. Ceci est dû au fait que le type SEQUENCE est un ensemble ordonné d'éléments, et que par le fait même, l'ordre est important. De plus, le type de la valeur peut donner une indication supplémentaire au sujet de l'élément désigné. La valeur TRUE, de type BOOLEAN, ne peut être que la valeur du dernier élément.

Un problème se pose cependant lorsque certains éléments sont optionnels ou peuvent prendre une valeur par défaut. Si on définit un type SEQUENCE {INTEGER OPTIONAL,INTEGER OPTIONAL,BOOLEAN}, il se peut que certaines notations de valeur soient ambiguës. En effet, on peut interpréter la valeur {123,TRUE} de deux manières différentes : 123 est la valeur soit du premier élément (le second étant omis), soit du second élément

(le premier étant absent). La valeur {123,321,TRUE} n'est cependant pas ambiguë.

Une définition de type de donnée peut faire surgir des ambiguïtés dans la notation de valeur lorsqu'il est impossible de déterminer grâce à l'ordre et aux types des éléments (donnés par la définition du type SEQUENCE), leur valeur respective. Ce cas se présente lorsque plusieurs éléments sont de même type et qu'au moins un d'entre eux est optionnel.

Il faut pour éviter cette situation, définir des noms de référence, qui interviennent dans la notation de valeur pour distinguer ces éléments. La spécification de l'exemple précédent doit donc être modifiée comme suit : SEQUENCE {employeeNumber INTEGER OPTIONAL,phoneNumber INTEGER OPTIONAL,BOOLEAN}. Les noms de référence se trouvent alors explicitement dans toute notation de valeur, comme par exemple : {phoneNumber 555031,TRUE}. Il est dès lors possible de déterminer sans ambiguïté la valeur de chacun des éléments.

La représentation standard d'une valeur du type SEQUENCE est un élément de donnée de la classe universelle, de forme constructeur et d'identificateur 16. Le composant V est constitué des représentations associées aux éléments de la SEQUENCE, disposés dans l'ordre indiqué par la définition. Lorsque des valeurs d'éléments ne sont pas présentes dans la notation de valeur, elles ne sont pas représentées dans le codage.

Le problème de l'ambiguïté peut également se poser pour le codage TLV. En effet les sous-éléments de donnée sont identifiés par leur position relative dans l'ensemble d'octets et par leur composant type.

En cas de litige (lorsque des éléments optionnels sont de mêmes types), il faut définir un nouvel identificateur de type pour ces éléments. Ceci se fera au moyen du TAGGED TYPE (point 3.3.2.2).

Un exemple de définition et de représentation concrète d'un type SEQUENCE est donnée au point 3.3.3.

Le type SET

Le type SET s'apparente fort au type SEQUENCE, en ce sens qu'il modélise également un ensemble constitué de plusieurs éléments. On parlera de membres d'un SET,

plutôt que d'éléments, pour les distinguer par rapport à la SEQUENCE.

Le SET se distingue par le fait que l'ordre des membres n'a aucune importance. On peut ainsi désigner les valeurs des éléments dans n'importe quel ordre.

La grammaire de notation du type SET et des valeurs correspondantes est la suivante :

```
SetType             ::= SET | SET OF Type |
                        SET {MemberTypes}

MemberTypes         ::= OptionalTypeList | empty

OptionalTypeList    ::= OptionalType |
                        OptionalTypeList , OptionalType

OptionalType        ::= NamedType |
                        NamedType OPTIONAL |
                        NamedType DEFAULT Value |
                        ComponentsOf

NamedType           ::= identifier Type | Type

ComponentsOf        ::= COMPONENTS OF Type


SetValue            ::= {MemberValues}

MemberValues        ::= NamedValueList | empty

NamedValueList      ::= NamedValue |
                        NamedValueList , NamedValue

NamedValue          ::= identifier Value | Value
```

Il se peut, tout comme pour le type SEQUENCE, que la notation des valeurs d'un type SET soit ambiguë. Il existe cependant une contrainte supplémentaire, étant donné que l'ordre dans lequel les valeurs des membres sont citées n'a rien de commun avec l'ordre de définition de ceux-ci. C'est pourquoi un nom de référence doit accompagner la définition des membres de même type, même si ceux-ci ne sont pas optionnels.

Les identificateurs de ces membres doivent également être distincts, étant donné que le codage TLV ne reflète pas non plus l'ordre de définition des membres. Ainsi, seul l'identificateur d'un membre du SET permet de repérer correctement le codage de sa valeur dans l'ensemble d'octets.

La représentation standard d'une valeur du type SET est un élément de donnée de la classe universelle, de forme constructeur et d'identificateur 17. Le composant V est constitué de zéro ou plusieurs sous-éléments, qui sont les représentations associées aux membres du SET. L'ordre d'apparition de ces sous-éléments dans la représentation n'est pas déterminé.

Un exemple de définition et de représentation concrète d'un type SET est donnée au point 3.3.3.

Le type CHOICE

Le type CHOICE permet de modéliser un type de données équivalent à un parmi plusieurs types possibles. La valeur du type CHOICE est une valeur du type choisi. On peut ainsi spécifier une donnée dont le type ne peut être déterminé que lorsque l'on connait la valeur.

La grammaire de notation du type CHOICE et des valeurs correspondantes est la suivante :

```
ChoiceType            ::= CHOICE {AlternativeTypeList}

AlternativeTypeList   ::= NamedType |
                          AlternativeTypeList ,
                          NamedType

NamedTyped            ::= identifier Type | Type

ChoiceValue           ::= identifier Value | Value
```

Un des types possibles évoqués dans le CHOICE peut être le type NULL, abordé précédemment. On peut ainsi modéliser une donnée qui quelques fois représente une information significative, et d'autre fois ne contient pas d'information. Ainsi, une valeur du type CHOICE {phoneNumber INTEGER,NULL} permet de représenter une information significative, dans le cas où le numéro de téléphone existe, et de signaler l'absence d'information dans le cas contraire.

Un cas particulier du type CHOICE est le type pré-défini ANY. Ce dernier est équivalent à un choix parmi tous les types de données pré-définis, et tous les types définis par l'utilisateur. Voici la grammaire de la notation standard du type ANY et d'une des valeur correspondantes :

```
AnyType      ::=   ANY

AnyValue     ::=   Type Value
```

On constate qu'il est nécessaire, pour dénoter une valeur de ANY, de spécifier son type, étant donné qu'on ne peut pas le déduire de la définition même de ANY.

Un exemple de définition et de représentation concrète d'un type CHOICE est donnée au point 3.3.3.

3.3.2.2. Définition de types de données

Le langage X409 permet de définir le format des informations qu'un système est susceptible d'échanger avec son environnement. On construit ces types de donnée soit à partir des types pré-définis, soit en se basant sur des types définis par l'utilisateur.

La grammaire de la notation introduisant une définition type ou la définition d'une de ses valeurs est la suivante :

```
TypeDefinition      ::=  identifier "::=" Type
ValueDefinition     ::=  identifier Type "::=" Value
```

On peut ainsi définir le type de données PrimaryColor comme étant équivalent à un type entier, pouvant prendre les valeurs 0, 1 et 2, qui portent respectivement les noms de référence red, yellow, blue :

```
PrimaryColor ::= INTEGER {red(0), yellow(1), blue(2)}
```

On dira que PrimaryColor est le type défini, tandis que INTEGER {red(0), yellow(1), blue(2)} est le type de base. La notation d'une valeur du type défini est identique à celle utilisée pour la notation d'une valeur du type de base.

Nous distinguerons les types de base élémentaires et constructeurs.

Un type de base est élémentaire lorsqu'il est un des types pré-définis primitifs (BOOLEAN, INTEGER, BIT STRING, OCTET STRING et NULL), ou lorsqu'il fait référence à un autre type défini dans la spécification.

Ainsi, les deux définitions suivantes ont des types de base constructeurs :

```
OtherColor      ::= PrimaryColor
PrimaryColor    ::= INTEGER {red(0), yellow(1), blue(2)}
```

Un type de base est dit constructeur lorsqu'il fait intervenir un des types pré-définis constructeurs (SEQUENCE, SET et CHOICE). Le type de base de la définition suivante est constructeur :

```
Example ::= SEQUENCE {INTEGER, BOOLEAN}
```

Il est également possible d'identifier une valeur particulière parmi toutes les valeurs possibles du type. Si l'on décide de spécifier qu'il existe une valeur par défaut du type PrimaryColor, on procédera de la manière suivante :

```
defaultColor PrimaryColor ::= yellow
```

Par convention, les identificateurs de types commencent par une lettre majuscule, tandis que les identificateurs de valeurs débutent par une minuscule. Ceci permet de distinguer tout de suite noms de types et noms de valeurs, sans devoir nécessairement consulter la définition.

La représentation standard d'une valeur du type défini est strictement la même que celle du type de base. Ceci implique qu'il est impossible de distinguer dans la représentation, tous les types définis dans la spécification. C'est la raison pour laquelle on introduit une construction supplémentaire, appelée Tagged Type, qui permet d'assigner un identificateur numérique d'une certaine classe au type que l'on définit. Celui-ci est codé dans le composant T de l'élément de donnée.

La grammaire associée à la notation du Tagged Type est la suivante :

```
TaggedType      ::= Tag IMPLICIT Type |
                    Tag Type

Tag             ::= [ Class number ]

Class           ::= UNIVERSAL | APPLICATION |
                    PRIVATE | empty
```

On assigne ainsi un nouvel identificateur en spécifiant sa valeur numérique (number) et sa classe (Class). Lorsque la classe n'est pas spécifiée explicitement, l'identificateur appartient par défaut à la classe CONTEXT.

Deux formes sont possibles pour le Tagged Type : l'identificateur peut être modifié de manière implicite ou de manière explicite. On indique le choix de la première forme en mentionnant IMPLICIT dans la définition. Lorsqu'on ne spécifie rien, la seconde forme est choisie.

Dans la première forme, seul le composant T est modifié : les bits cc sont remplacés par la valeur correspondant à la classe spécifiée, et les bits i prennent la valeur du nouvel identificateur. Les autres informations ne sont pas modifiées : le bit f du composant T est inchangé, et les composants L et V sont les mêmes que pour le codage d'une valeur du type de base. Il n'est donc plus possible de déterminer, à partir du codage TLV, l'identificateur et la classe du type de base.

Lorsqu'on définit le type Priority ::= [APPLICATION 7] IMPLICIT INTEGER, une valeur de ce type est par exemple l'entier 10. Elle est codée de la même manière qu'une valeur du type INTEGER, si ce n'est que les bits cc prennent la valeur correspondant à APPLICATION, et que les bits i valent 7.

Voici le codage de la valeur :

	Interprétation	Codage hexadécimal	Codage binaire
T	Priority [APPLICATION 7] IMPLICIT INTEGER	$(67)_{16}$	$(01\ 0\ 00111)_2$ APPLICATION PRIMITIVE 7
L	1	$(01)_{16}$	$(0\ 0000001)_2$ longueur sur un octet : 1
V	10	$(0A)_{16}$	$(0000\ 1010)_2$

Il n'est donc plus possible de déterminer à partir du codage TLV que le type de base du type Priority est un INTEGER.

Pour la forme explicite, le codage est différent. En effet, le composant T contient dans les bits cc et les bits i les valeurs correspondant respectivement à la classe et à l'identificateur spécifiés. Le bit f vaut 1 et signale un codage sous forme constructeur : le composant V contient en effet l'élément de donnée correspondant à la valeur du type de base.

Lorsqu'on définit le type Priority ::= [APPLICATION 7] INTEGER, une valeur de ce type est par exemple l'entier 10. Elle est codée de la manière suivante :

	Interprétation		Codage hexadécimal	Codage binaire
T	Priority [APPLICATION 7] INTEGER		$(67)_{16}$	$(01\ 1\ 00111)_2$ — APPLICATION, CONSTR., 7
L	3		$(03)_{16}$	$(0\ 0000011)_2$ — longueur sur un octet, 3
V	T	INTEGER	$(02)_{16}$	$(00\ 0\ 00010)_2$ — UNIVERSAL, PRIMITIVE, 2
	L	1	$(01)_{16}$	$(0\ 0000001)_2$ — longueur sur un octet, 1
	V	10	$(0A)_{16}$	$(0000\ 1010)_2$

On constate que le codage TLV révèle de manière explicite que le type Priority est construit à partir du type INTEGER.

Il est important de noter que la composante TaggedType établit un lien explicite avec une représentation physique particulière. Le langage ne peut donc plus être considéré comme abstrait. Cependant, le retrait de cette composante introduit la définition d'un nouveau langage, qui lui sera abstrait. Il suffit alors de remplacer les règles de grammaire associées aux Tagged Type par la règle suivante :

TaggedType ::= Type

La recommandation X409 définit un certain nombre de types standards. Ils ne sont pas des types pré-définis du langage mais représentent des types de donnée fréquemment utilisés. Voici deux de ces types, définissant

respectivement une chaîne de caractères ASCII, et une chaîne de caractères représentant la date et l'heure :

```
IA5String  ::= [UNIVERSAL 22] IMPLICIT OCTET STRING
Time       ::= [UNIVERSAL 23] IMPLICIT OCTET STRING
```

On constate que ces types ont un identificateur de la classe universelle, étant donné qu'ils sont susceptibles d'intervenir dans un grand nombre de domaines d'application.

3.3.2.3. Modification des notations standards

La recommandation X409 propose un langage intégrant de notations de types de données et des valeurs correspondantes. Ce langage permet également de définir des nouvelles notations, appelées notations non-standards.

Celles-ci sont introduites à l'aide de la construction MACRO. Elle définit simplement de nouvelles règles de grammaire exprimant les nouvelles conventions de notation. Elles seront simplement ajoutées à la grammaire standard.

Nous ne détaillerons pas ici les règles de grammaires associées à l'écriture de telles MACROS. Le lecteur désireux d'obtenir de plus amples informations à ce sujet, peut se référer à [24].

3.3.2.4. Structuration des définitions

Les définitions de types de données qui appartiennent à la même spécification sont regroupées dans un module. Celui-ci est défini par une notation respectant les règles suivantes :

```
ModuleDefinition   ::=  identifier DEFINITIONS "::="
                        BEGIN ModuleBody END

ModuleBody         ::=  DefinitionList | empty

DefinitionList     ::=  DefinitionList Definition

Definition         ::=  TypeDefinition |
                        ValueDefinition |
                        MacroDefinition
```

On peut ainsi par exemple structurer la spécification des couleurs, présentée au point 3.3.2.2, comme suit :

```
Color DEFINITIONS ::=
BEGIN
  PrimaryColor                ::=  INTEGER
                                   {red(0),yellow(1),blue(2)}
  defaultColor PrimaryColor   ::= yellow
END
```

3.3.3. Exemple

Le langage X409 défini plus haut a été utilisé la première fois dans le cadre des recommandations de la série X400 sur la messagerie électronique. Nous présenterons dans ce qui suit un extrait de la spécification des données échangées entre deux noeuds de relais de messages (MTAs). Nous avons vu (cf. chapitre 2) que l'information véhiculée est constituée d'une enveloppe contenant les indications nécessaires pour que le relais s'opère correctement, et d'un contenu qui est le message émis par l'utilisateur.

Nous ne considérerons ici que quelques éléments des données échangées, de manière à mettre en évidence les concepts abordés dans les points 3.3.1 et 3.3.2. Nous donnerons successivement la spécification X409 des types de données, et le codage TLV pour une valeur de ces types.

```
P1 DEFINITIONS ::=
BEGIN

  MPDU           ::= CHOICE {[0] IMPLICIT UserMPDU,
                             ServiceMPDU}
  UserMPDU       ::= SEQUENCE {UMPDUEnvelope,UMPDUContent}
  UMPDUEnvelope  ::= SET {
                         ...
                         Priority DEFAULT normal,
                         ...
                         deferredDelivery [0]
                             IMPLICIT Time OPTIONAL,
                         ...
                         }
  UMPDUContent   ::= OCTET STRING
  Priority       ::= [APPLICATION 7] IMPLICIT
                         INTEGER {normal(0),nonUrgent(1),
                                  urgent(2)}

  ...

END
```

La spécification P1 peut être interprétée comme suit. L'unité de donnée de protocole (MPDU) peut représenter deux choses : il est soit relatif au relais de message (UserMPDU), soit utilisé pour véhiculer de l'information de service entre les deux noeuds MTA (ServiceMPDU). Le UserMPDU est constitué d'une enveloppe (UMPDUEnvelope) et d'un contenu (UMPDUContent). Le type du composant contenu n'est pas spécifié de manière plus fine étant donné qu'il véhicule de l'information à caractère "privé", et qui n'intervient nullement au niveau du relais.

L'enveloppe contient un ensemble d'informations nécessaires au relais du message. Nous n'évoquerons que les éléments spécifiant la priorité (Priority) à accorder au message, et l'heure avant laquelle celui-ci ne peut en aucun cas être fourni à l'utilisateur final (deferredDelivery Time).

L'élément "Priority" est un entier dont on distingue plusieurs valeurs, identifiées par des noms de référence (normal, nonUrgent, urgent). Il faut noter ici l'utilisation d'un tag de la classe application. Ceci signifie que l'on accorde une importance particulière au type Priority, dans le cadre de cette spécification.

Il faut noter que l'élément "Priority" peut éventuellement être omis dans une instance particulière de l'enveloppe, sa valeur étant assumée par défaut à "normal".

Le champ relatif à la fourniture différée a été désigné de manière explicite et porte le nom de référence "deferredDelivery". Le type Time est un des types standards proposés dans la recommandation X409, et ne doit pas être redéfini.

On peut coder la valeur { { ..., urgent, ..., deferredDelivery "12h30", ...} } comme suit :

				Interprétation	Codage hexadécimal	Codage binaire
T				MPDU [0] IMPLICIT UserMPDU	$(A0)_{16}$	$(10\ 1\ 00000)_2$ CONTEXT, CONSTR., 0
L				...	$(...)_{16}$	$(...)_2$
V	T			UMPDU Envelope	$(31)_{16}$	$(00\ 1\ 10001)_2$ UNIVERSAL, CONSTR., 17
	L			...	$(...)_{16}$	$(...)_2$
				...		
	V	T		Priority	$(67)_{16}$	$(01\ 1\ 00111)_2$ APPLICATION, PRIM., 7
		L		1	$(01)_{16}$	$(0\ 0000001)_2$ longueur sur un octet, 1
		V		urgent	$(02)_{16}$	$(0\ 0000010)_2$
				...		
	V	T		Priority [0] IMPLICIT Time	$(80)_{16}$	$(10\ 0\ 00000)_2$ CONTEXT, PRIM., 0
		L		5	$(05)_{16}$	$(0\ 0000101)_2$ longueur sur un octet, 5
		V		"12h30"	$(31\ 32\ 68\ 33\ 30)_{16}$	$(0011\ 0001\ 0011\ 0010\ 0110\ 1000\ 0011\ 0011\ 0011\ 0000)_2$

3.3.4. Evaluation

Nous proposerons dans ce qui suit une évaluation du langage X409 sur base des critères énoncés au point 3.2.

Le langage X409 est certainement applicable à la spécification de systèmes communiquants. Il se limite cependant à la modélisation des données échangées par de tels systèmes avec leur environnement. Ce langage revêt une importance non négligeable, étant donné qu'il est susceptible d'intervenir dans la spécification des unités de donnée des protocoles de la couche application.

Le langage a déjà été utilisé pour la spécification des unités de donnée des protocoles P1 [26], P2 [27] et P3 [25] de la messagerie X400. Le format standard de document SFD (Simple Formattable Document) a également été défini grâce à ce langage [27].

Une autre architecture de document est proposée dans la recommandation T73 [7]. Elle est également décrite au moyen du langage X409.

Le langage permet de modéliser assez facilement les informations échangées par un système avec son environnement. Il est possible d'exprimer certaines contraintes sémantiques (un entier ne peut par exemple que prendre les valeurs indiquées par les noms de référence énoncés), d'autres contraintes ne sont pas modélisables, et doivent être ajoutées en commentaire. On ne peut pas spécifier dans le langage que le type "PackedBCDString" est une suite d'octets correspondant au codage binaire condensé de nombres décimaux. Il n'est en effet pas possible de spécifier que ces octets ne peuvent prendre que certaines valeurs. La définition qui en est donnée est la suivante [24] :

```
PackedBCDString ::= OCTET STRING
        -- the digits 0 through 9, two digits per octet
        -- (1111)2 used for padding
```

Certains types de donnée importants ne sont pas disponibles dans le langage, comme par exemple le type "nombre réel". Il est cependant probable que le langage sera complété dans les versions ultérieures [32].

Le langage X409 est relativement indépendant des décisions d'implémentation, si on lui retire la composante "Tagged Type", qui établit un lien explicite avec un codage physique particulier.

Notons également qu'il est possible de faire abstraction des données que l'on ne désire pas spécifier. Ainsi, dans la spécification des unités de donnée du protocole P1, on ne détaille pas plus loin le contenu du message, qui n'a pas de signification pour les noeuds de relais. On le considère simplement comme une suite d'octets :

```
UMPDUContent ::= OCTET STRING
```

Le contenu du message est alors décrit de manière plus fine dans la spécification des unités de donnée du protocole P2, puisque pour les UAs, le contenu est significatif.

Le langage X409 est facilement communicable, étant donné la simplicité de ses constructions et le nombre restreint de types de base à connaître.

Un outil a déjà été développé pour le langage X409 [29]. Il permet de vérifier que la spécification respecte bien les conventions du langage énoncées dans la recommandation X409. D'autre part, il génère un programme d'application permettant de réaliser des tests de conformité. On peut ainsi vérifier qu'une unité de donnée de protocole émise par l'implémentation à tester respecte bien les conventions du codage TLV, et la visualiser sous forme arborescente, de manière à mettre en évidence la structure hiérarchique du codage. D'autre part, il est possible de composer une unité de donnée de protocole à émettre vers l'implémentation à tester. Ces unités peuvent également être générées automatiquement à partir de la spécification donnée.

3.4. Langages formels de spécification du comportement

Nous aborderons dans cette partie les langages formels permettant de modéliser le comportement externe d'un système. Nous présenterons brièvement deux langages en voie de normalisation : ESTELLE et LOTOS.

Le langage ESTELLE permet également de spécifier l'aspect des données échangées avec l'environnement. LOTOS ne le permet pas directement. On peut cependant intégrer dans une spécification LOTOS des déclarations utilisant un formalisme défini dans un autre langage (ACT ONE) pour spécifier l'aspect des données.

Cette présentation est brève et incomplète, étant donné que le sujet des chapitres suivants est essentiellement l'utilisation pratique du langage X409 de définition des données. Elle met seulement en évidence les aspects généraux

et les concepts de base utilisés dans les langages ESTELLE et LOTOS. Nous n'aborderons pas non plus la syntaxe tel que nous l'avons fait pour le langage X409.

Nous présenterons au point 3.4.3 la spécification d'un protocole de la couche transport rédigée dans les langages ESTELLE et LOTOS.

Le lecteur désirant une information plus détaillée peut consulter les références [3], [10] et [11].

3.4.1. Le langage ESTELLE

ESTELLE est l'un des langages formels de spécification en voie de normalisation auprès de l'organisme international ISO [3]. Nous verrons successivement les principes de structuration, les techniques de description utilisées pour la spécification d'un sous-système, et enfin les modalités de communication entre les différents éléments du système.

3.4.1.1. Principes de structuration

Un système peut se structurer en un certain nombre de sous-systèmes communiquant via des portes d'interaction (figure 3.8). Cette décomposition peut s'appliquer récursivement aux sous-systèmes ainsi obtenus.

L'environnement du système est constitué de ses utilisateurs. Si le système complet est le modèle de référence OSI (cf chapitre 1), l'environnement comprend par exemple les programmes d'application qui utilisent les sept couches pour dialoguer avec des applications distantes. L'environnement d'un sous-système est le système qui l'englobe.

Les portes d'interaction permettent le dialogue entre un sous-système et son environnement. Chaque sous-système peut disposer de plusieurs portes. En effet, un "père" peut connecter dynamiquement une de ses propres portes à une porte d'un de ses "fils", ou relier certaines portes de ses "fils" entre elles. L'effet de cette connexion est la redirection des interactions transitant par la première porte, vers la seconde.

La figure 3.8 montre une connexion de ce genre. Le système S0 connecte dynamiquement les portes P1 et P4 de ses fils (S1 et S2). Ceci a comme conséquence que les interactions émises par S1 vers son environnement (S0) sont redirigées vers la porte P4. Le sous-système S2

reçoit alors ces interactions comme si elles émanaient de son environnement (S0).

Il est clair que l'on ne désire pas toujours rediriger toutes les interactions. C'est pourquoi il est possible de définir plusieurs portes. Ainsi le sous-système S1 dispose de la porte P2 pour émettre des interactions vers son "père" (S0).

Chacun des sous-systèmes est modélisé par un automate à états finis susceptible de réagir aux stimuli de l'environnement, transmis par le biais de ces portes.

La spécification du système porte essentiellement sur la subdivision en sous-systèmes, la définition des modalités d'interaction entre ceux-ci, et la description des états et transitions de chacun des automates.

ESTELLE permet au spécificateur de déterminer le niveau de précision à atteindre. Il pourra par exemple choisir de définir complètement un sous-système, en allant jusqu'à la décomposition en sous-systèmes élémentaires, ou de n'en spécifier que les portes d'interaction, considérant ainsi le sous-système comme une boîte noire.

Lorsque l'on désire par exemple spécifier le modèle OSI, on peut choisir de ne spécifier que son interface avec l'environnement. On ne détaille donc pas la structuration en couches.

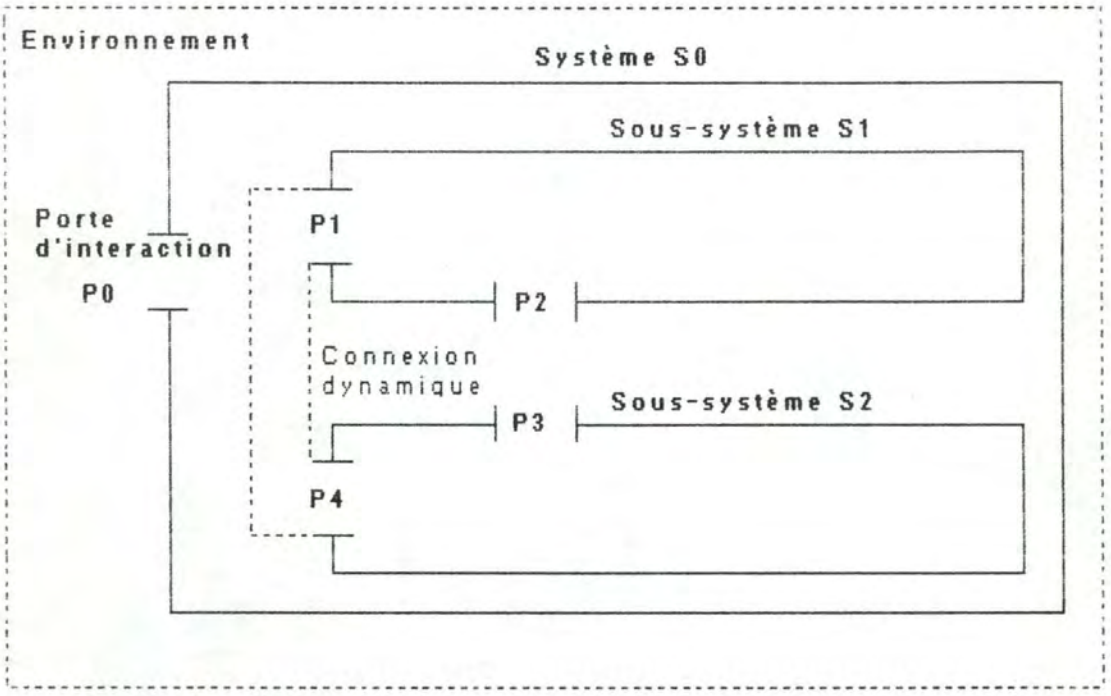


Figure 3.8

Principes de structuration en ESTELLE

La structure hiérarchique des sous-systèmes est exprimée de manière statique dans la spécification. La position du texte source des sous-systèmes dans la spécification complète du système qui les englobe, exprime la position relative du sous-système dans la hiérarchie. Cette hiérarchie est donc statique.

ESTELLE intègre également une composante dynamique, en permettant à un sous-système "père" de créer ou supprimer des instances de sous-systèmes "fils", de niveau juste inférieur de la hiérarchie.

3.4.1.2. Automates à états finis

Le comportement d'un sous-système est spécifié par l'intermédiaire d'un automate à états finis. Celui-ci est constitué d'un ensemble d'états, la transition d'un état à l'autre étant définie comme réaction à un stimulus fourni par l'environnement. Certaines transitions peuvent être spontanées, dans le cas où elles ont lieu sans qu'elles soient provoquées par l'environnement.

En ESTELLE on peut exprimer quel stimulus est susceptible de provoquer une transition, quel doit être l'état courant et quel sera le suivant, et quelles actions

doivent être entreprises lors du passage d'un état à l'autre. En plus, il est possible de spécifier quelles conditions doivent être vérifiées pour que la transition puisse être exécutée.

Les actions que l'automate doit exécuter sont décrites par l'intermédiaire de blocs d'instructions dans un langage de programmation. Il s'agit du PASCAL ISO agrémenté d'un certain nombre d'instructions propres à ESTELLE (production d'interactions aux portes, gestion dynamique des sous-systèmes, ...).

On peut ici aussi envisager plusieurs niveaux de détail dans la spécification. On spécifie par exemple complètement les actions, en programmant explicitement tous les détails, ou bien on se limite à une découpe plus grossière en ne définissant que l'agencement des différentes sous-tâches. Les sous-tâches sont alors considérées comme des primitives que l'on suppose sous-jacentes.

Les sous-systèmes ainsi définis peuvent soit fonctionner de manière parallèle, soit s'exécuter suivant des modalités séquentielles. Ceci signifie que dans le cas de deux sous-systèmes parallèles, lorsqu'un automate désire exécuter une transition, il peut le faire indépendamment des transitions de l'autre automate. Dans le cas de deux automates non parallèles offrant chacun une transition à exécuter, une des deux transitions seulement sera choisie et exécutée effectivement. Ce choix s'effectue de manière non déterministe.

3.4.1.3. Communication entre sous-systèmes

Deux mécanismes permettent en ESTELLE la communication entre sous-systèmes : l'échange de messages par le biais de portes d'interaction et l'exportation de valeurs de variables.

Les portes d'interaction sont des canaux bidirectionnels de communication entre sous-systèmes. On spécifie le rôle joué par chacun des participants, et le type d'interaction possible pour chaque rôle, accompagné de la description en PASCAL de la structure des données échangées. Ces portes sont des files d'attente, où les événements reçus sont mémorisés dans un ordre FIFO.

Les automates modélisant les sous-systèmes peuvent se référer explicitement aux interactions reçues via ces portes. C'est ainsi qu'un automate pourra exécuter une transition lorsqu'un certain événement sera détecté sur une de ses portes d'interaction. Un automate peut en

revanche émettre un événement à destination de l'environnement.

Comme nous l'avons vu plus haut, les portes d'interaction de chacun des sous-systèmes peuvent être dynamiquement reliées entre elles par un sous-système "père".

Un sous-système "père" dispose également d'un autre moyen de communication. Il a en effet accès aux valeurs des variables des sous-systèmes "fils". Ces valeurs sont donc exportées d'un sous-système vers le sous-système qui l'englobe.

3.4.1.4. Evaluation

Nous proposerons dans ce qui suit une évaluation du langage ESTELLE sur base des critères énoncés au point 3.2.

Le langage ESTELLE permet une modélisation aisée de systèmes communiquants, étant donné ses possibilités de structuration. A chaque étape de la spécification, on peut faire abstraction des autres sous-systèmes, qui sont alors considérés comme des "boîtes noires".

Le langage permet une modélisation aisée du comportement des sous-systèmes, qui est essentiellement basée sur une représentation par automates à états finis. Cette forme de description est souvent utilisée dans les documents normalisés. Le passage de cette représentation à la spécification ESTELLE pour un protocole donné est donc assez immédiat .

Le langage ESTELLE est cependant doté d'un ensemble de caractéristiques qui font qu'une spécification intègre trop de détails techniques d'implémentation. Le spécificateur a bien sûr la possibilité de laisser certains de ces choix en suspens, mais ceci entache évidemment la qualité et la précision de la spécification. Ce lien trop étroit avec des questions relevant de l'implémentation se situe au niveau de la description des données et la définition des actions associées aux transitions des automates. Dans cette description, on utilise nécessairement les concepts de base du langage PASCAL, ce qui empêche par exemple de retarder le choix d'un langage de programmation ou d'une structure de données particulière jusqu'à la phase d'implémentation du logiciel.

Il faut cependant noter que les spécifications ESTELLE peuvent être rédigées de manière très claire. Ceci est principalement dû à la modélisation du comportement par un automate à états finis, approche avec laquelle on est plus familiarisé, puisque ce mode de représentation a été fréquemment utilisé dans les documents officiels de normalisation.

D'autre part, le spécificateur pourra disposer dans un proche avenir d'un environnement logiciel d'aide à la spécification [6]. A ce moment-là, il est nettement plus intéressant de disposer d'un langage proche du langage de programmation qui sera utilisé lors de l'implémentation. On pourra en effet tout d'abord spécifier le système en n'explicitant que les automates, valider cette approche, pour ensuite la raffiner. On s'approcherait ainsi de plus en plus d'une implémentation du système, tout en s'assurant de la validité des différentes étapes que l'on franchit.

Un exemple d'utilisation du langage ESTELLE concernant la spécification d'un protocole de la couche transport se trouve au point 3.4.3.

3.4.2. Le langage LOTOS

Le langage LOTOS est, tout comme le langage ESTELLE, en voie de normalisation au niveau international. Il permet également la spécification du comportement d'un système, tel qu'il peut être observé à partir de l'environnement. Il possède cependant un ensemble de caractéristiques qui le distinguent d'ESTELLE. Nous insisterons ici sur ces caractéristiques spécifiques.

Le lecteur désirant de plus amples informations peut consulter les références [12], [14] et [15].

3.4.2.1. Expressions de comportement

Le comportement d'un sous-système à spécifier est décrit en toute généralité par une expression de comportement. Elle s'exprime en fonction des événements qui peuvent survenir à l'interface avec l'environnement, et décrit l'enchaînement temporel de ceux-ci.

Ces expressions peuvent modéliser des comportements très divers : enchaînement séquentiel, parallélisme, exclusion mutuelle, avortement, ... Toutes ces formes ont

pour base les expressions élémentaires de comportement, spécifiant l'ordonnancement temporel des événements.

L'enchaînement séquentiel permet de rendre équivalent l'expression du comportement global, à l'enchaînement de deux comportements plus élémentaires. On pourra ainsi dire que le système se comporte comme décrit dans la première sous-expression, puis se comporte comme décrit dans la deuxième.

Le parallélisme entre deux expressions de comportement décrit que le comportement global est équivalent aux comportements réunis de deux sous-systèmes coexistants. Différentes formes de parallélisme sont possibles : soit les deux sous-systèmes n'ont pas besoin d'interagir, auquel cas aucune synchronisation n'est imposée, soit ils doivent se synchroniser à un moment donné, pour pouvoir continuer leur évolution. Les formes de synchronisation, que nous ne détaillerons pas ici, sont très riches et ne se limitent pas à la simple émission/réception de messages.

L'exclusion mutuelle permet d'exprimer que le système se comporte comme l'un de ses sous-systèmes, mais un seul seulement. Le choix entre l'une ou l'autre possibilité de comportement peut s'effectuer d'une manière déterministe ou non déterministe, sur base de la présence d'événements à l'interface, ou être non-déterministe.

L'avortement exprime que le comportement d'un système est d'abord le comportement d'un premier sous-système, puis, lorsqu'un deuxième sous-système se trouve déclenché, le premier est tout simplement ignoré. Le comportement global devient alors le comportement du deuxième sous-système.

Il faut noter qu'une spécification LOTOS a pour objectif principal de spécifier l'ordonnancement des événements à l'interface avec l'environnement. Ceci implique que les "traitements" à effectuer pour provoquer de tels événements, ou pour répondre à ceux-ci, ne sont spécifiés nulle part. Il est cependant possible de les décrire dans une fonction dont le résultat est un événement, et qui effectue les traitements nécessaires. Cet aspect n'est cependant pas primordial.

3.4.2.2. Spécification des données

Les données qu'un système échange avec son environnement sont des événements possédant un type et des valeurs. La description des types n'est pas explicitement

prévue en LOTOS. On intègre pour ce faire un autre langage, ACT ONE, qui permet la définition de types abstraits.

3.4.2.3. Evaluation

Nous proposerons dans ce qui suit une évaluation du langage LOTOS sur base des critères énoncés au point 3.2.

Le langage LOTOS est facilement applicable à la spécification de systèmes communiquants, au même titre qu'ESTELLE. Des exemples d'utilisation du langage LOTOS dans le cadre du modèle OSI sont présentés dans [13], [14] et [15].

LOTOS possède cependant un désavantage majeur : le formalisme qu'il utilise est relativement complexe, ce qui peut impliquer certaines difficultés de compréhension de la spécification par une personne étrangère.

Le langage LOTOS, agrémenté du langage de spécification des données ACT ONE, peut être considéré comme totalement indépendant par rapport à des choix d'implémentation. Il n'intègre en effet aucune composante qui permettrait d'établir un lien explicite avec une implémentation particulière.

Etant donné la richesse et la concision de ses expressions, le langage LOTOS nous semble peu communicable. Il est en effet nécessaire d'ajouter une portion significative de commentaires en langue naturelle pour faciliter la compréhension.

Pour LOTOS, il n'existe pas encore d'environnement d'aide à la spécification. Le langage possède cependant des bases mathématiques solides offrant des possibilités de preuve formelle. Leduc dans [5] démontre ainsi que la spécification LOTOS d'un protocole transport jointe à la spécification du service réseau sous-jacent, offre bien le service transport attendu. Cette démonstration est possible parce que le langage possède les bases théoriques nécessaires.

Un exemple d'utilisation du langage LOTOS concernant la spécification d'un protocole de la couche transport se trouve au point 3.4.3.

3.4.3. Exemple

Nous allons présenter dans ce qui suit un exemple de spécification du comportement d'un système distribué, rédigée dans les langages ESTELLE et LOTOS.

Il s'agit de l'Alternating Bit Protocol, qui régit le dialogue entre deux entités de la couche transport. Celles-ci fournissent aux entités session un service de transport de messages fiable, en utilisant un service réseau plus ou moins fiable. Il se peut qu'occasionnellement des messages soient perdus par l'entité réseau.

Nous ne considérons ici que la partie du protocole consacrée à la phase d'échange d'information. Nous supposons que la connexion entre les entités session est déjà établie.

Les primitives de service disponibles aux entités session et transport sont représentées schématiquement à la figure 3.9.

Les primitives que peut utiliser l'entité session sont les suivantes : SEND_req (Send request) pour émettre un message vers l'entité paire et RECEIVE_req (Receive request) pour demander un message reçu de cette entité. L'entité transport peut répondre à cette dernière primitive par l'intermédiaire d'un RECEIVE_rsp (RECEIVE response), en fournissant à l'entité session l'information reçue.

L'entité transport peut émettre un message à l'aide de la primitive DATA_req (Data request). L'entité réseau lui signale l'arrivée d'un message par le biais de la primitive DATA_rsp (DATA response).

Etant donné que le service réseau n'est pas fiable, et que de temps en temps des messages sont perdus, chaque information échangée entre les entités transport est accompagnée d'un numéro de séquence.

La perte des messages est cependant rare, et ainsi un numéro de séquence variant entre zéro et un suffit.

Chaque entité mémorise un numéro de séquence pour les messages émis et les messages reçus. En début de communication, ces compteurs sont initialisés à zéro.

Lorsqu'une entité transport émet un message, elle y ajoute le numéro de séquence d'émission. Après l'envoi, elle

attend un accusé de réception signalant que l'entité paire a bien reçu le message.

L'entité transport paire, lorsqu'elle reçoit un message, vérifie si le numéro de séquence qu'il contient correspond à son numéro de séquence de réception. Si les deux valeurs sont identiques, elle envoie un accusé de réception positif et augmente son compteur de réception de 1 modulo 2, après avoir mémorisé l'information dans un buffer. Celle-ci est ensuite disponible pour l'entité session, qui signale sa demande d'accès par RECEIVE_req. Dans le cas contraire, le message portant le numéro de séquence de réception a été perdu. L'entité transport émet alors un accusé de réception négatif et laisse le compteur inchangé.

Lorsqu'un accusé de réception positif parvient à l'entité qui a émis un message, celle-ci incrémente son numéro de séquence d'émission de 1 modulo 2. Si l'accusé de réception est négatif, ou s'il ne lui parvient pas endéans un certain temps (timeout), l'entité ne modifie pas son numéro de séquence d'émission, et retransmet le message comme décrit plus haut.

Les entités paires de la couche transport ne peuvent donc émettre qu'un seul message à la fois, étant donné qu'elles doivent attendre l'accusé de réception de l'entité paire. Elles peuvent cependant chacune assumer en même temps le rôle d'émetteur et de récepteur.

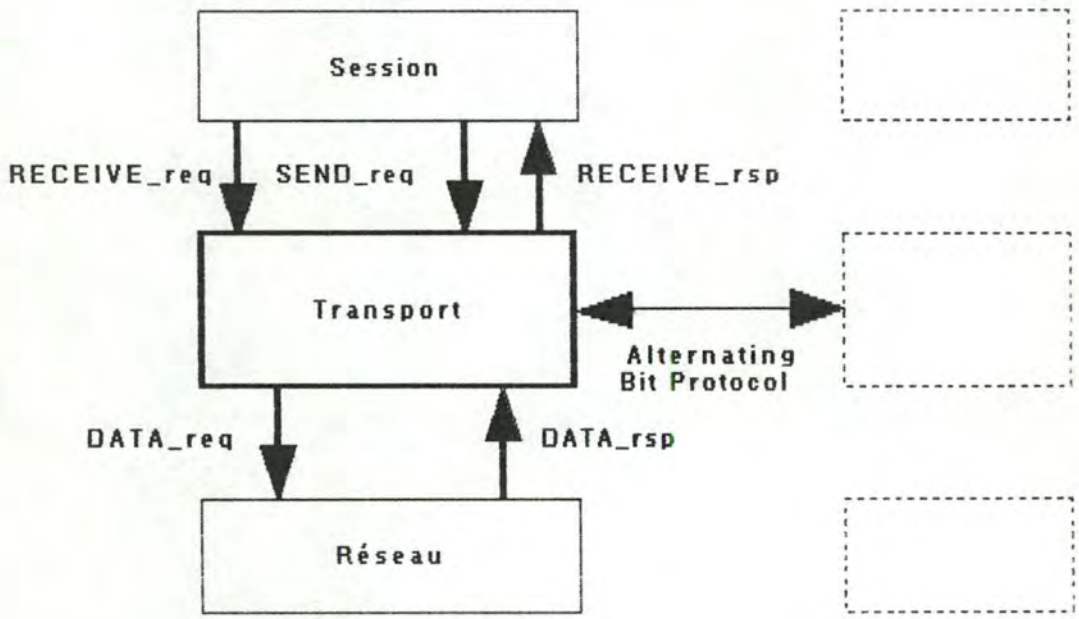


Figure 3.9

Alternating bit protocol

On peut représenter le comportement d'une entité transport par un automate à états finis (figure 3.10). Cette modélisation est déduite directement de la spécification informelle donnée plus haut.

L'automate possède deux états. Le premier (Established ou ESTAB) indique que la connexion est établie et que l'entité transport est en attente des primitives de service émanant des deux couches adjacentes. Le second état (Acknowledgement Wait ou ACK_WAIT) représente l'attente d'un accusé de réception de l'entité paire.

Les transitions de la figure 3.10 sont représentées à l'aide d'arcs orientés et mentionnent l'état initial, l'événement déclencheur et l'état final. On ne détaille pas pour chacune d'elles les opérations que doit exécuter l'entité transport au moment de la transition. Celles-ci peuvent être aisément déduites de la présentation informelle donnée plus haut.

De l'état ESTAB, on peut passer à l'état ACK_WAIT lorsque l'entité session invoque une primitive de service SEND_req. On retourne à l'état ESTAB lorsque l'on reçoit de l'entité réseau une primitive DATA_rsp signalant l'arrivée d'un accusé de réception positif de l'entité paire.

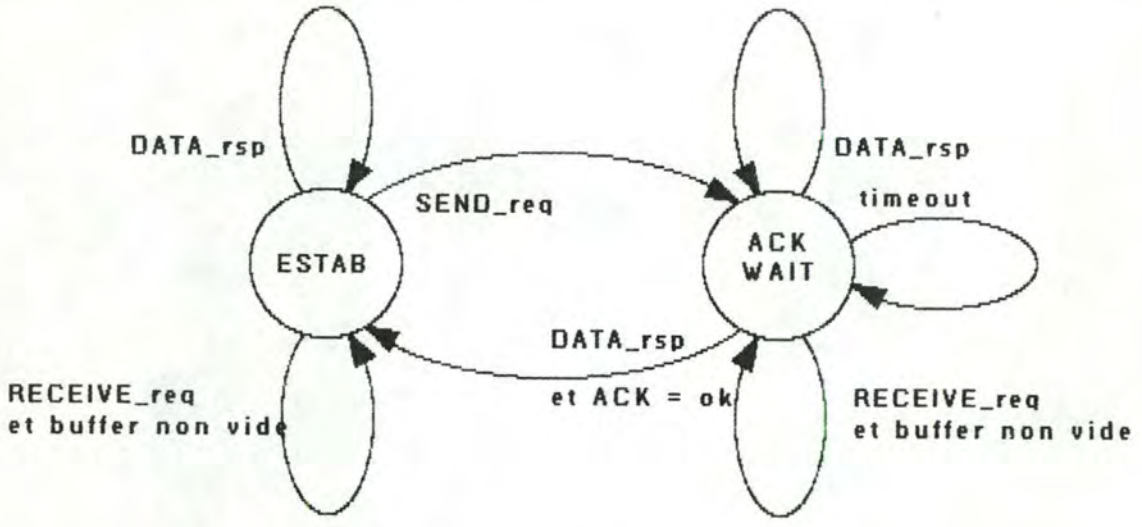


Figure 3.10

Représentation par automate à états finis
pour l'Alternating Bit Protocol

On exécute une transition vers le même état (ESTAB ou ACK_WAIT) lorsque l'entité session émet une primitive

RECEIVE_req et qu'on peut lui fournir un message (buffer non vide), ou bien lorsque l'entité réseau signale par l'intermédiaire d'un DATA_rsp l'arrivée d'un message de type "données" émanant de l'entité paire.

On exécute une transition de l'état ACK_WAIT vers lui-même lorsque l'accusé de réception n'a pas été reçu endéans une limite de temps fixée (timeout).

3.4.3.1. Spécification ESTELLE

La spécification complète en ESTELLE de l'exemple cité plus haut se trouve dans [3]. Nous n'en présenterons ici qu'un extrait.

```
Specification Alternating_Bit_Protocol

...

        channel U_acces_point
(Service_User,Service_Provider);
             by Service_User : SEND_req (...);
                                  RECEIVE_req;
             by Service_Provider : RECEIVE_rsp (...);

        channel N_acces_point
(Service_User,Service_Provider);
             by Service_User : DATA_req (...);
             by Service_Provider : DATA_rsp (...);

...


module User_type process;
                 ip U : U_acces_point (Service_User);
end;

module Alternating_Bit_type process;
                 ip U : U_acces_point (Service_Provider);
                    N : N_acces_point (Service_User);
end;

module Network_type process;
                 ip N : N_acces_point (Service_Provider);
end;

body User_Body for User_type; external;

body Network_Body for Network_type; external;

body Alternating_Bit_Body for Alternating_Bit_type

        state : ACK_WAIT,ESTAB;
```

```
statset EITHER = [ACK_WAIT,ESTAB];

...
initialize
     to ESTAB
          begin

               initialisation des compteurs
                    d'émission
                    et de réception

          end;

trans
     from ESTAB
          to ACK_WAIT
               when U.SEND_req (...)
                    begin
                       construction du message en
                       ajoutant le compteur
                         d'émission

                       output N.DATA_req (...);

                    end


trans
     from EITHER
          to same
               when U.RECEIVE_req (...)
               provided "buffer non vide"

                    begin

                         extraction du message

                         output
                          U.RECEIVE_rsp (...);

                    end


trans
     from ACK_WAIT
          to ACK_WAIT
               delay (timeout)
                    begin

                         construction du message à
                              retransmettre

                         output N.DATA_req (...);

                    end

trans
     from ACK_WAIT
          to ESTAB
               when N.DATA_rsp (...)
```

```
                    provided ACK = ok
                        begin
                            incrémentation du
                                compteur
                                d'émission

                        end

        trans
            from EITHER
                to same
                    when N.DATA_rsp (...)
                    provided "message de données"

                        begin

                            copie du message dans
                                le buffer

                            émission du ACK
                                positif ou
                                négatif
                            output
                              N.DATA_req (...);

                            incrémentation du
                            compteur de réception
                                si nécessaire
                        end
```

```
...

modvar Session : User_type;
       Transport : Alternating_Bit_type;
       Network : Network_type;


initialize
        begin

            init Session with User_body;
            init Transport with Alternating_Bit_body;
            init Network with Network_body;

            connect Session.U to Transport.U;
            connect Transport.N to Network.N;
        end

end.
```

On définit tout d'abord les types de points d'interaction pouvant exister au sein du système. On

déclare par exemple le type U_access_point, dont les deux utilisateurs sont Service_Provider et Service_User. Le premier utilisateur dispose des primitives SEND_req et RECEIVE_req, le second de la primitive RECEIVE_rsp. Ceci est défini au moyen de l'instruction channel.

Pour chaque sous-système, on définit une entête et un corps. L'entête contient la description de l'interface avec l'environnement, tandis que le corps contient la définition du comportement observable.

L'entête de l'entité transport est introduite par l'instruction module, et porte le nom Alternating_Bit_type. On y déclare entre autre le point d'interaction U (vers le sous-système User) qui est de type U_access_point. Le sous-système transport assume pour ce point d'interaction le rôle de fournisseur du service (Service_Provider), et dispose donc de la primitive RECEIVE_req décrite dans la spécification du type de point d'interaction. Le point d'interaction N (vers le sous-système Network) est déclaré de la même manière.

Le corps de l'entité transport porte le nom Alternating_Bit_body et est introduit par l'instruction body. On y fait référence à l'entête définie précédemment (for Alternating_Bit_type). Ce corps contient la description du comportement du sous-système modélisé par un automate à états finis.

On notera que les corps des deux autres sous-systèmes (User_body et Network_body) ne sont pas détaillés. Le mot-clé external indique qu'ils sont définis dans une autre spécification.

Les états ACK_WAIT et ESTAB de l'automate modélisant le comportement de l'entité transport sont déclarés au moyen de l'instruction state. On déclare également l'ensemble EITHER constitué des deux états. Nous en verrons plus loin l'utilisation.

On spécifie l'état initial de l'automate par l'instruction initialize to ESTAB. Les actions à exécuter lors de cette initialisation sont décrites dans un bloc d'instructions délimité par begin et end.

Les transitions sont définies au moyen de l'instruction trans. On indique l'état initial (from), l'état final (to), l'événement déclencheur (when) et la condition de déclenchement (provided). Les actions à exécuter lors de cette transition sont décrites dans un bloc d'instruction délimité par begin et end.

Ainsi, trans from ESTAB to ACK_WAIT when U.SEND_req begin ... end définit la transition de l'état ESTAB vers l'état ACK_WAIT, à exécuter lorsque l'entité transport reçoit une primitive SEND_req par le biais de la porte U. Notons dans les actions l'instruction output N.DATA_req qui émet une primitive DATA_req vers l'environnement par le biais de la porte N.

Il est possible, lorsque deux transitions impliquent les mêmes actions et les mêmes événements déclencheurs, de condenser leurs déclarations. C'est le cas lors de la réception d'une primitive RECEIVE_req sur la porte U. Que l'automate soit dans l'état ESTAB ou dans l'état ACK_WAIT, les actions à exécuter sont identiques. L'état final est l'état courant de l'automate avant la transition. Ceci est spécifié par l'instruction trans from EITHER to same. L'état initial de cette transition est un des deux états de l'ensemble EITHER, l'état final étant l'état initial. On notera ici également la déclaration d'une condition de déclenchement (provided "buffer non vide").

Il est possible de spécifier un temps d'attente pour l'exécution d'une transition. Si aucune autre transition ne peut être prise en compte endéans un certain temps, et si l'état courant est l'état initial spécifié, alors cette transition est exécutée. Ceci est spécifié par le mot-clé delay (timeout).

Le corps du système complet contient les déclarations des variables relatives aux sous-systèmes, et une phase d'initialisation.

On y déclare une variable pour chaque sous-système par l'instruction modvar. Ainsi la variable Transport est déclarée de type Alternating_Bit_type via l'instruction modvar Transport : Alternating_Bit_type.

La phase d'initialisation est un bloc d'instructions délimité par begin et end. On y trouve des instructions de création dynamique d'instances de sous-systèmes et de connexion dynamique des portes d'interaction. Ainsi, init Session with User_body a comme effet de créer une instance d'un sous-système, qui peut par la suite être désignée par la variable Session, et dont le comportement est celui spécifié dans User_body.

Les portes U des instances de sous-systèmes Session et Transport sont connectées dynamiquement par l'instruction connect Session.U to Transport.U. Ceci a comme effet de rediriger toutes les interactions émises par Session via sa porte U, vers la porte U de Transport, et vice-versa.

3.4.3.2. Spécification LOTOS

```
Specification Alternating_Bit_Protocol
...

process Whole_System :=
       ((Session [U] | [U] | Transport [U,N])
                       | [N] | Network [N])

  where
       process Session [U] :=
            ...
       endproc

       process Network [U] :=
            ...
       endproc

       process Transport [U,N] :=

            Estab [U,N]

            where process Estab [U,N] :=

                      (U ? SEND_req (...);
                           N ! DATA_req (...);
                                Ack_wait [U,N])
                 []
                      (N ? DATA_rsp (...);Estab [U,N])
                 []
                      ([buffer non vide]
                                -> U ? RECEIVE_req(...);
                                     U ! RECEIVE_rsp(...);
                                          Estab [U,N])

                 where process Ack_wait [U,N] :=

                           (i;N ! DATA_req (...);
                                Ack_wait [U,N])
                      []
                           ([buffer non vide]
                                 -> U ? RECEIVE_req(...);
                                      U ! RECEIVE_rsp(...);
                                           Ack_wait [U,N])
                      []
                           (N ? DATA_rsp (...);
                                (([données ou ack négatif]
                                ->   N ! DATA_req(...);
                                          Ack_wait [U,N])
                                []
                                ([ack positif]
                                     ->   Estab [U,N]))

                 endproc
            endproc
       endproc
  endproc
...
endspec
```

La spécification Alternating_Bit_Protocol se comporte comme le système Whole_System. Ce comportement est déclaré par le biais de l'instruction process Whole_System := ... endproc.

L'expression indique que Whole_System est composé de trois sous-systèmes (Session, Transport et Network). Session dispose d'une porte d'interaction U (Session [U]), Transport possède deux portes U et N (Transport [U,N]) et Network utilise la porte N (Network [N]).

Les sous-systèmes Session et Transport s'exécutent en parallèle, tout en se synchronisant sur les interactions transitant par la porte U. Ceci est indiqué par l'expression de comportement Session [U] | [U] | Transport [U,N]. Ces deux sous-systèmes s'exécutent en parallèle avec Network, en se synchronisant sur les interactions transitant par la porte U. Ceci est déclaré par l'expression de comportement (Session [U] | [U] | Transport [U,N]) | [U] | Network [U].

Le comportement observable de Whole_System est simplement équivalent aux comportements réunis des trois sous-systèmes.

Les deux sous-systèmes Session et Network ne sont pas détaillés plus loin.

Le sous-système Transport est défini comme ayant le même comportement que Estab, par l'instruction process Transport [U,N] := Estab [U,N]. On constate qu'il est possible de passer en paramètre des noms de portes d'interaction.

Le sous-système Estab dispose donc des portes U et N. Son comportement est défini par process Estab := ... endproc, et choisi parmi trois comportements possibles. Ceci est spécifié par l'opérateur [].

Le premier comportement cité est observable lorsque le sous-système Estab peut se synchroniser avec l'environnement sur une primitive SEND_req transitant par le biais de la porte U. U ? SEND_req (...) indique que le sous-système peut se synchroniser avec un autre sous-système (Session) émettant U ! SEND_req (...). Cette synchronisation correspond à l'émission/réception d'une primitive de service sur une porte d'interaction. Lorsque cette synchronisation a lieu, le comportement observable du sous-système Estab est défini par l'expression N ! DATA_req;Ack_wait [U,N]. Ceci signifie que l'entité transport, après avoir reçu une primitive SEND_req de l'entité session, émet un DATA_req vers l'entité réseau,

et se comporte ensuite comme le sous-système Ack_wait [U,N].

Le second comportement peut être interprété de la même manière. On constate ici que le sous-système Estab [U,N], suite à la réception d'un DATA_rsp sur la porte N, se comporte comme le sous-système $\overline{\text{E}}$stab [U,N]. Il est donc possible de définir un comportement infini de manière récursive.

Le troisième comportement que peut avoir le sous-système Estab est observable lorsqu'il reçoit sur la porte U une primitive RECEIVE_req et qu'une condition est vérifié (buffer non vide).

Un comportement est choisi effectivement parmi les trois comportements évoqués, suivant les événements disponibles à l'interface avec l'environnement. Lorsque plusieurs synchronisations sont possibles, une parmi celles-ci est choisie de manière non-déterministe.

Le sous-système Ack_wait [U,N] est défini de manière similaire. On notera ici la présence d'un événement particulier i (événement interne). Celui-ci peut avoir lieu à n'importe quel moment sans intervention de l'environnement. Il modélise dans cette spécification le fait que de temps en temps, l'accusé de réception n'est pas reçu endéans un certain temps (timeout). Lorsque cette expression de comportement est choisie parmi les trois possibles, le sous-système Ack_wait émet une primitive DATA_req vers la porte N pour retransmettre le message supposé perdu. Ensuite le sous-système se comporte comme Ack_wait [U,N]. On retrouve ici la description récursive d'un comportement infini.

3.5. Conclusion

Les différentes caractéristiques évoquées pour chaque langage formel sont résumées à la figure 3.11. Les critères énoncés précédemment sont repris, et pour chaque langage, on trouve une appréciation du respect de chaque critère. Cette appréciation peut être très bonne (++), bonne (+), négative (-) ou très mauvaise (--).

Langage / Critère	X 409	ESTELLE	LOTOS
Domaine	données	traitements et données	traitements
Applicabilité au modèle OSI	+	++	++
Facilité d'utilisation	+	++	-
Possibilité d'abstraction	-	--	++
Puissance d'expression	+	+	++
Disponibilité d'outils	+	+	+

Figure 3.11

Evaluation des différents langages

4. Outil d'analyse et de production d'unités de données de protocole

4.1. Introduction

Nous avons vu au chapitre 3 certains critères permettant d'évaluer les qualités des langages formels de spécification. La disponibilité d'outils de vérification du texte formel et de génération automatique de code est un critère important. Il détermine en effet l'utilité du langage formel dans la conception et surtout la réalisation d'applications informatiques.

Nous présenterons dans ce qui suit un outil susceptible d'être utilisé par des programmes de la couche application du modèle OSI, manipulant des unités de donnée de protocoles spécifiées par le biais du langage X409. Il doit permettre à ces applications d'utiliser pour le stockage local d'informations, une représentation plus adéquate que le codage TLV. Il doit également fournir une translation bidirectionnelle entre ce format local et la représentation proposée par la recommandation X409.

Lors de la réalisation de cet outil d'analyse et de construction d'unités de donnée de protocole, il est important de veiller à ce qu'il soit applicable à n'importe quelle spécification. Au chapitre 5, nous montrerons comment mettre en oeuvre les fonctionnalités proposées, dans le cadre d'une spécification particulière. Le chapitre 6 propose alors un second outil, permettant la génération automatique de l'outil d'analyse et de production d'unités de donnée de protocole à partir d'une spécification quelconque. Il vérifie la cohérence du texte de spécification et génère automatiquement l'outil d'interface proposé ici.

Nous mettrons en évidence au point 4.2 les avantages et inconvénients du codage TLV, justifiant d'une part son utilisation pour l'échange d'informations entre des entités paires, et d'autre part la réalisation d'un tel outil.

Nous présenterons au point 4.3 les fonctionnalités que doit assumer l'outil proposé. Les domaines potentiels d'application de celui-ci seront évoqués au point 4.4.

4.2. Avantages et inconvénients du codage TLV

Le codage TLV est le format proposé par la recommandation X409 pour représenter les valeurs abstraites des types de donnée définis par le biais du langage X409.

Ce format est un moyen très simple, permettant d'exprimer des valeurs de longueurs quelconques. Cette simplicité est essentiellement dûe aux caractéristiques du langage. On peut en effet spécifier n'importe quel type de donnée en fonction d'un nombre restreint de types pré-définis. Il en est de même pour la représentation TLV. Il suffit de maîtriser les conventions de représentation des valeurs des types pré-définis, et les principes de codage des composants Type, Longueur et Valeur, pour pouvoir représenter une valeur d'un des types définis dans la spécification.

Le codage TLV est tout-à-fait général, en ce sens qu'il peut être utilisé pour représenter les valeurs de tous les types de données susceptibles d'être définis par le biais du langage X409.

Le codage proposé par la recommandation X409 est un format adéquat pour la transmission de données. En effet, une valeur abstraite quelconque est toujours représentée par une suite d'octets consécutifs que l'on peut aisément transmettre sur une ligne de communication. Cet ensemble intègre à la fois des "informations utiles" (dans les composants V élémentaires) et des indications quant à la structure de ces informations (dans les composants T et L).

Les propositions de la recommandation X409 ont une portée plus générale que le cadre de la messagerie électronique. En effet, le langage X409 est susceptible d'intervenir dans la spécification des unités de donnée des protocoles de toute application de la couche sept du modèle OSI. Il permet de décrire les syntaxes abstraites définissant les caractéristiques sémantiques des données échangées entre les entités paires de cette couche. Le codage TLV intervient alors en tant que syntaxe concrète, définissant le format réel des informations échangées entre les entités présentation.

Une remarque s'impose sur l'utilisation de ces propositions dans le cadre de la messagerie électronique. En effet, les recommandations de la série X400 imposent le codage TLV en tant que syntaxe concrète. Ceci signifie que les entités présentation n'ont pas le droit de négocier une autre syntaxe concrète lors de l'ouverture d'une communication entre les entités application. Ces recommandations vont même plus loin, en imposant aux entités application d'émettre les informations dans le format TLV.

Ces obligations ont avant tout deux conséquences. D'une part, elles rendent la couche présentation inutile. En effet, les entités ne peuvent pas négocier de syntaxe concrète, étant donné que celle-ci est imposée. De plus, les entités application sont obligées d'émettre les informations sous forme de codage TLV. Il n'est donc plus nécessaire d'établir une correspondance entre un format d'émission choisi par une entité application et la syntaxe concrète utilisée par les entités présentation. D'autre part, ces obligations mettent en cause la philosophie générale du modèle OSI, à savoir qu'il est nécessaire que les applications soient libres de choisir une représentation propre pour l'échange d'information avec une application distante.

Le codage TLV est cependant une représentation possédant certains points faibles. En effet, la proportion d'informations utiles dans l'ensemble d'octets représentant une valeur est assez faible. Les exemples exposés au chapitre 3 sont assez révélateurs. Le codage de la valeur TRUE du type BOOLEAN nécessite trois octets (point 3.3.2.1). La représentation est encore plus gourmande dans le cas d'un Tagged Type de forme explicite (point 3.3.2.2).

De plus, la représentation TLV ne permet qu'un accès séquentiel à l'information. Il est en effet nécessaire de décoder les composants Type et Longueur d'un élément de donnée avant de pouvoir extraire le composant Valeur et interpréter correctement l'information qu'il contient.

Dans l'exemple proposé au point 3.3.3, on constate que l'accès au membre deferredDelivery n'est pas immédiat. On doit en effet décoder le premier composant Type pour déterminer que la valeur du MPDU est une valeur de type UserMPDU. Le premier composant Longueur permet l'accès au composant V. Celui-ci est composé de plusieurs éléments de donnée (UMPDUEnvelope et UMPDUContent).

L'analyse du premier de ces éléments de donnée indique qu'il s'agit bien d'une valeur du type UMPDUEnvelope. Le composant L permet de déterminer la position relative du composant V, qui est également constitué de plusieurs éléments de donnée. Pour accéder à l'élément de donnée correspondant à deferredDelivery, il est nécessaire de parcourir séquentiellement le composant V, étant donné qu'il représente une valeur d'un type SET. En effet, la position relative du codage de la valeur du membre deferredDelivery n'est pas nécessairement celle donnée dans la spécification du type SET.

On obtient finalement l'élément de donnée correspondant au membre cherché. Après avoir décodé les composants T et L, on peut enfin déterminer que la valeur du membre deferredDelivery est "12h30".

L'accès à une information élémentaire est relativement fastidieux, étant donné la structure hiérarchique du codage TLV. Celle-ci découle directement de la définition hiérarchique des types de donnée, inhérente au langage X409

Il est également très difficile de modifier une information élémentaire au sein de l'ensemble d'octets. Si l'on veut remplacer dans le codage de l'exemple présenté au point 3.3.3, le composant V de l'élément de donnée associé à deferredDelivery par une autre valeur, plus longue ou plus courte, il est nécessaire de modifier tous les composants L situés avant cet élément de donnée.

Ceci montre bien que le codage TLV, tout en étant approprié pour représenter des informations échangées sur une ligne de communication, n'est certainement pas une structure de donnée exploitable au sein d'un programme d'application.

Un programme d'application peut donc choisir un autre format pour représenter les valeurs abstraites. Celui-ci serait par exemple une structure de donnée disponible dans un langage de programmation. L'application peut ainsi disposer pour le traitement local d'une représentation commode et efficace, et lors de la transmission vers une application distante, produire la représentation TLV pour la valeur de la structure de donnée que l'on désire transmettre.

Elle pourrait procéder de même pour les informations qu'elle reçoit de l'application distante. La valeur reçue serait tout simplement convertie, de manière à être représentée conformément à la structure de donnée choisie.

Cette façon de procéder est explicitement autorisée dans le cadre du modèle OSI, étant donné qu'une application est libre de choisir une structure quelconque pour le stockage local d'informations. Elle doit seulement respecter certaines conventions lors du dialogue avec l'entité paire.

L'interface entre les deux représentations peut être réalisé par un module spécifique, venant tout simplement s'insérer dans les logiciels d'application.

Une entité application exploite donc la structure de donnée locale. Lorsqu'elle désire émettre l'information qui y est stockée vers l'entité paire, elle invoque simplement les services de ce module. Celui-ci produit alors l'unité de donnée de protocole dans le format TLV. L'ensemble d'octets produit est ensuite émis par l'entité application vers l'entité paire.

De même, lorsqu'une entité application reçoit une unité de donnée de protocole de l'entité paire, elle invoque les services du module. Celui-ci analyse alors l'unité reçue, vérifie si elle respecte les conventions du codage TLV et si elle représente une valeur possible d'un des types de donnée définis dans la spécification. Il garnit ensuite la structure de donnée avec l'information contenue dans l'unité de donnée de protocole.

Il est important de noter que ce module dépend fortement de la spécification des informations que les applications peuvent échanger. En effet, le codage TLV n'est rien d'autre qu'une représentation concrète d'une valeur particulière d'un des types définis par le biais du langage X409. Ainsi, le module doit être à même d'encoder et de décoder des valeurs pour tous les types définis dans une spécification particulière. Lorsque la spécification change, le module doit également être mis à jour.

Ce module doit donc être conçu de manière à pouvoir être rapidement adapté à une nouvelle spécification. On peut également envisager, comme nous le ferons au chapitre 6, la génération automatique de ce module à partir du texte de la spécification.

4.3. Spécification de l'outil d'analyse et de production d'unités de données de protocoles

Nous avons vu au point 4.2 qu'il est préférable que les programmes d'application disposent d'une représentation plus commode et plus efficace que le codage TLV pour le stockage local des informations. L'interface entre le codage normalisé et la représentation locale est réalisée par un module d'analyse et de production d'unités de donnée de protocole.

On peut envisager deux fonctionnalités supplémentaires : d'une part la saisie d'une notation abstraite de valeur et sa représentation dans la structure de données, et d'autre part la production de la notation abstraite pour une certaine valeur de la structure de données. Ces deux fonctionnalités peuvent s'avérer utiles, comme nous le verrons au point 4.4, lors de la mise au point d'une application ou dans le cadre des tests de conformité d'implémentations. Ceux-ci visent à établir si l'implémentation d'une couche du modèle OSI, proposée par un constructeur, respecte bien les standards en vigueur.

Les fonctionnalités du module sont représentées schématiquement à la figure 4.1.

Nous allons maintenant passer en revue ces fonctionnalités. Il est important de noter que l'on considère que la spécification en langage X409 est sous-jacente. Comme nous l'avons vu plus haut, le module dépend fortement de cette spécification. De plus, la structure de donnée utilisée par le programme d'application, le codage TLV et la notation de valeur ne sont que des représentations différentes de l'information décrite dans la spécification. Lorsque la spécification est modifiée, l'information décrite change, et donc également ses représentations possibles. Le module permettant de passer d'une représentation à l'autre est alors également modifié.

Nous appellerons ce module dans ce qui suit "module de transcodage", étant donné qu'il permet de traduire l'information représentée d'une certaine manière dans un autre format.

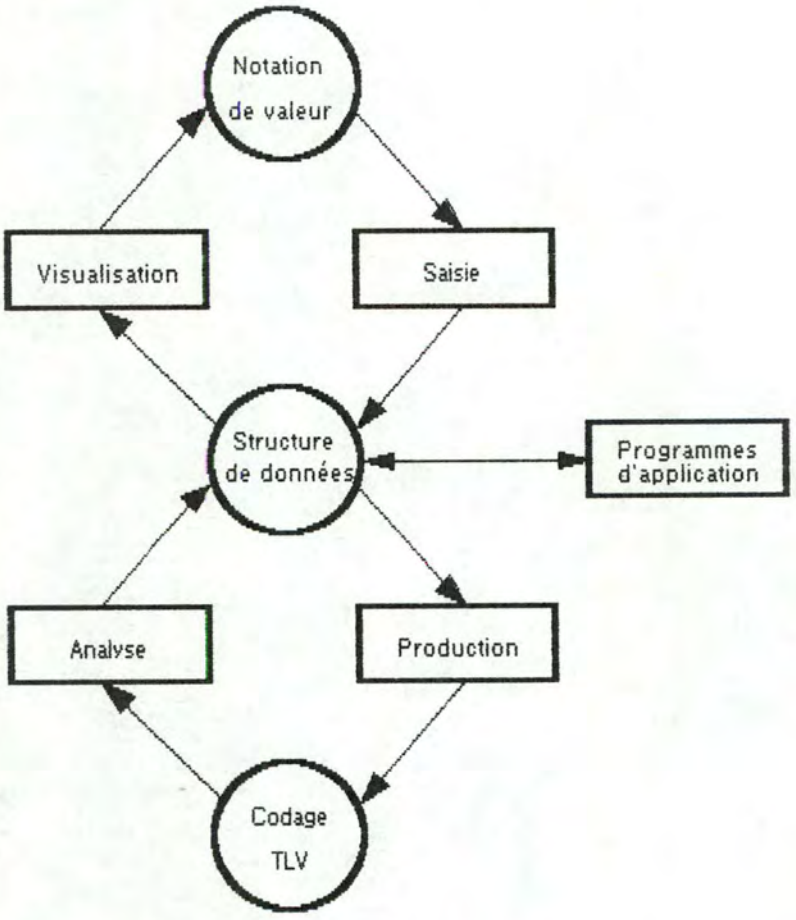


Figure 4.1

Fonctionnalités du module
d'analyse et de production
d'unités de données de protocoles

4.3.1. Fonctionnalité de production

Le module de transcodage doit permettre aux programmes d'application qui l'utilisent la production d'une suite d'octets consécutifs, représentant le codage TLV d'une valeur de la structure de donnée.

La structure de donnée est susceptible de représenter une valeur de n'importe quel type de donnée défini dans la spécification. Le module doit donc pouvoir produire le codage TLV pour une valeur de n'importe quel type défini.

Le codage TLV produit doit respecter les règles de codage énoncées au point 3.3.1.

4.3.2. Fonctionnalité d'analyse

Le module de transcodage doit permettre aux programmes d'application qui l'utilisent l'analyse d'une suite d'octets consécutifs, représentant le codage TLV d'une valeur d'un des types de donnée définis dans la spécification.

Il doit établir si le codage est conforme aux règles énoncées au point 3.3.1 et vérifier si la valeur codée est une valeur possible d'un des types définis dans la spécification.

Lorsque le codage est jugé correct, l'information qu'il contient doit être transférée dans la structure de donnée.

4.3.3. Fonctionnalité de visualisation

Le module de transcodage doit permettre aux programmes d'application qui l'utilisent la visualisation sous forme de notation de valeur, de l'information représentée dans la structure de donnée. La notation de valeur respecte les règles énoncées au point 3.3.2.

Le module doit être à même de visualiser la notation de valeur pour une valeur de n'importe quel type de donnée défini dans la spécification.

4.3.4. Fonctionnalité de saisie

Le module de transcodage doit permettre aux programmes d'application qui l'utilisent la saisie d'une valeur d'un des types définis dans la spécification. Il doit vérifier si celle-ci respecte les règles énoncées au point 3.3.2, et si elle dénote une valeur possible d'un des types définis.

Le module effectue ensuite les transformations nécessaires pour représenter la valeur dans la structure de donnée utilisée par le programme d'application.

Le module doit être à même de saisir une valeur pour n'importe quel type de donnée défini dans la spécification.

4.4. Domaines potentiels d'utilisation de l'outil d'analyse et de production d'unités de donnée de protocoles

Le module d'analyse et de production d'unités de donnée de protocole, ou module de transcodage, peut s'avérer utile dans plusieurs domaines d'application.

Une fois de plus, il faut noter que cet outil est tributaire de la spécification des unités de donnée de protocole, rédigée dans le langage X409.

4.4.1. Programmes d'application de la couche sept du modèle OSI

Le module de transcodage peut intervenir à plusieurs niveaux dans le développement et l'utilisation d'applications de la couche sept du modèle OSI.

Comme le montre la figure 4.2, les entités application peuvent utiliser les services du module de transcodage pour l'émission d'informations vers l'entité paire et la réception de données de cette entité.

Une entité application peut donc utiliser pour le traitement local, la structure de donnée choisie, et lors de l'émission, invoquer la fonctionnalité de production du module. Celui-ci transforme les informations reçues, de manière à respecter le codage TLV utilisé pour le dialogue entre les entités paires.

L'opération inverse est effectuée lors de la réception d'informations. Le module analyse le codage TLV reçu et garnit la structure de donnée de l'entité application. Celle-ci dispose alors de l'information émise par l'entité paire.

Dans le cadre de la messagerie électronique, exposée au chapitre 2, l'utilisation du module de transcodage est multiple. En effet, on distingue plusieurs protocoles entre les entités paires de la couche application. Pour chacun de ceux-ci, les entités disposent d'un module de transcodage spécifique au protocole utilisé. La figure 4.3 montre l'utilisation des modules spécifiques aux protocoles P1 et P2, utilisés respectivement par les entités MTAE et UAE. L'utilisation d'un module spécifique au protocole P3 par les entités SDE et MTAE est similaire.

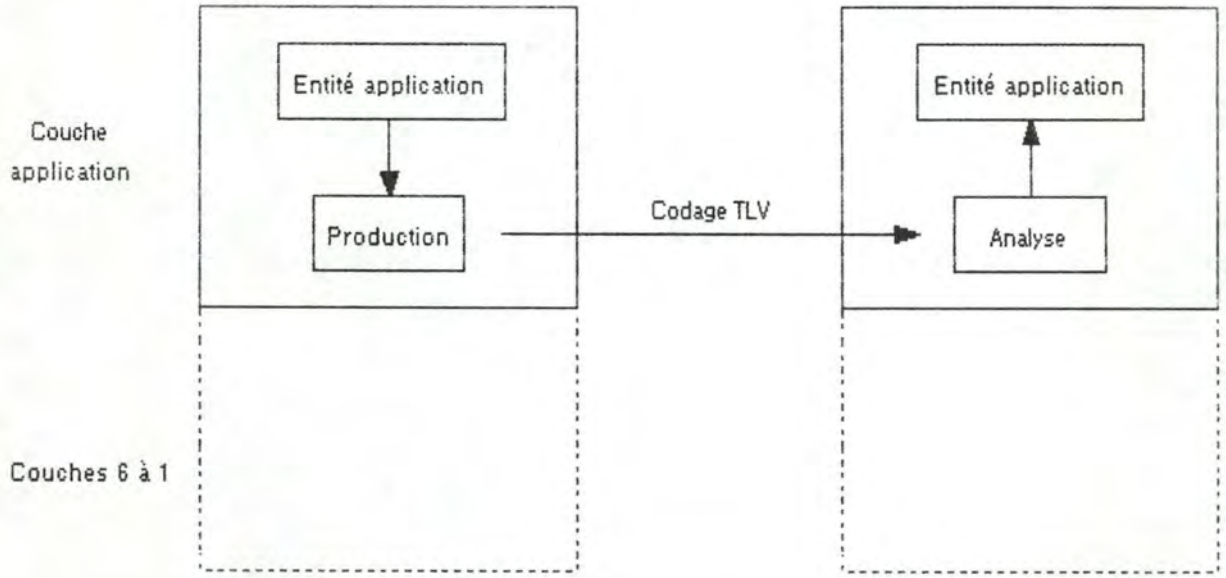


Figure 4.2

Utilisation du module de transcodage par une entité application

Le module de transcodage peut également intervenir dans le cadre d'autres protocoles, définissant les modalités de dialogue entre les entités utilisant les services de la couche UAL. Ce sera le cas pour la messagerie VMS, exposée au chapitre 2.

Le langage X409, comme nous l'avons vu, peut intervenir lors de la spécification d'une application pour décrire les données qu'elle échange avec son environnement. Celui-ci est par exemple une entité paire, mais peut également désigner un support externe de mémorisation.

On peut ainsi spécifier un format de document multimédia (cf. chapitre 2) par le biais du langage X409. Etant donné que ce document peut faire l'objet d'une mémorisation sur un support externe, le module de transcodage peut intervenir lors de l'échange d'informations avec ce support. Le codage TLV est alors utilisé comme format de représentation, et les applications peuvent disposer de cette information en invoquant la fonctionnalité d'analyse. De même, lors d'une demande de mémorisation d'information, l'application utilise le service de production, qui mémorise alors sur le support, le codage TLV relatif à cette information.

Bien entendu, le codage TLV n'est pas un format idéal pour la représentation d'informations sur un support externe. Cependant, le module de transcodage peut servir d'outil de développement dans le cadre de la mise au point d'une application. L'interface réalisé par ce module peut être ensuite remplacé par un module plus performant, utilisant un autre format de représentation.

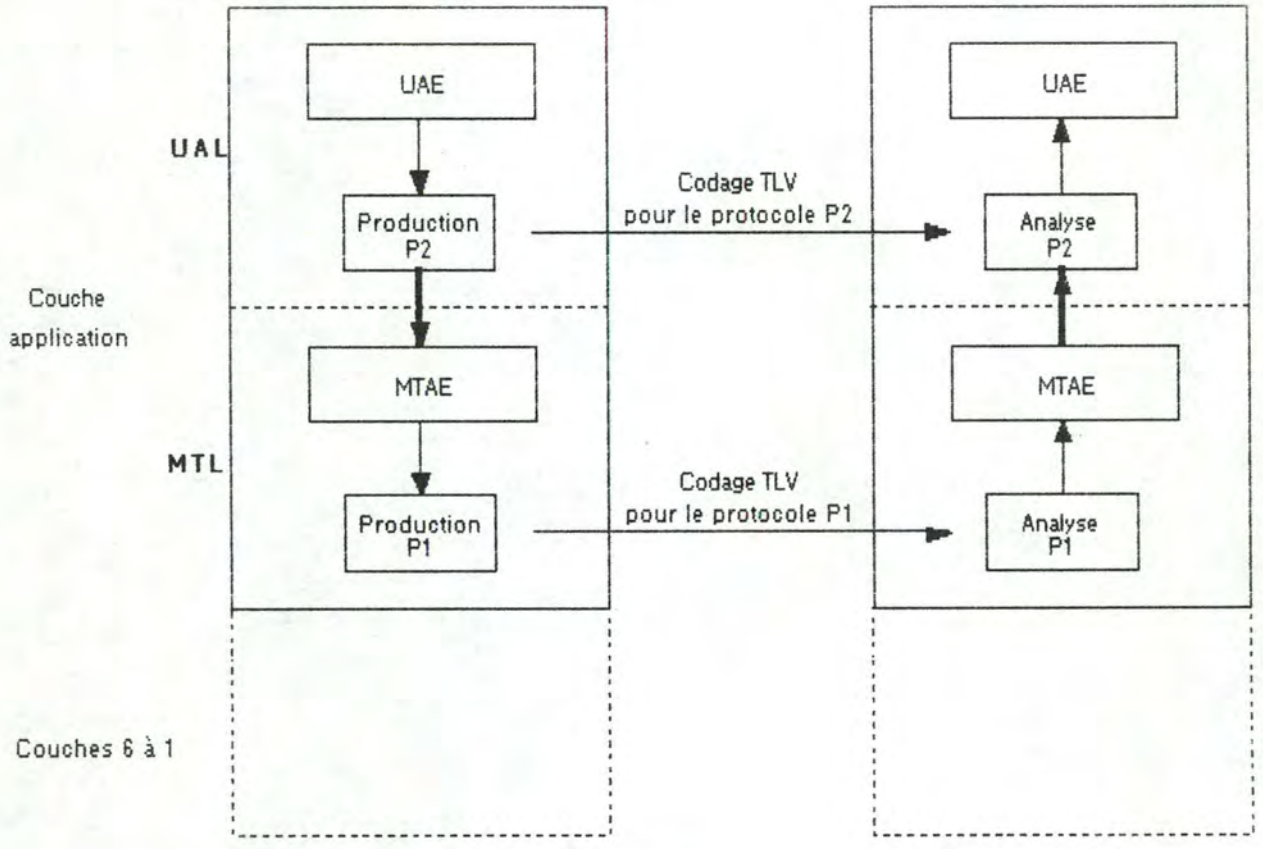


Figure 4.3

Utilisation du module de transcodage
par les entités de la messagerie électronique

Les fonctionnalités de visualisation et de saisie peuvent être utiles lors du développement d'une application de la couche sept du modèle OSI. La personne chargée de la mise au point ou des tests de cette implémentation dispose des fonctionnalités de visualisation et de saisie. Elle peut afficher la valeur de la structure de données utilisée par l'application, ou y stocker une nouvelle valeur. Elle peut également visualiser une unité de donnée de protocole reçue de l'entité paire, ou spécifier une unité à émettre vers cette entité

4.4.2. Tests de conformité d'implémentations

Les tests de conformité d'implémentations sont un autre domaine d'application du module de transcodage. Ils visent à établir si l'implémentation d'une couche du modèle OSI, proposée par un constructeur, respecte bien les standards en vigueur.

Les tests sont réalisés par un organisme indépendant, appelé centre de test. Il interagit avec l'implémentation à tester (IAT) située sur le site client, par l'intermédiaire d'un réseau de communication [4].

Différentes architectures sont présentées dans [4]. L'architecture du National Physical Laboratory (NPL) est représentée à la figure 4.4

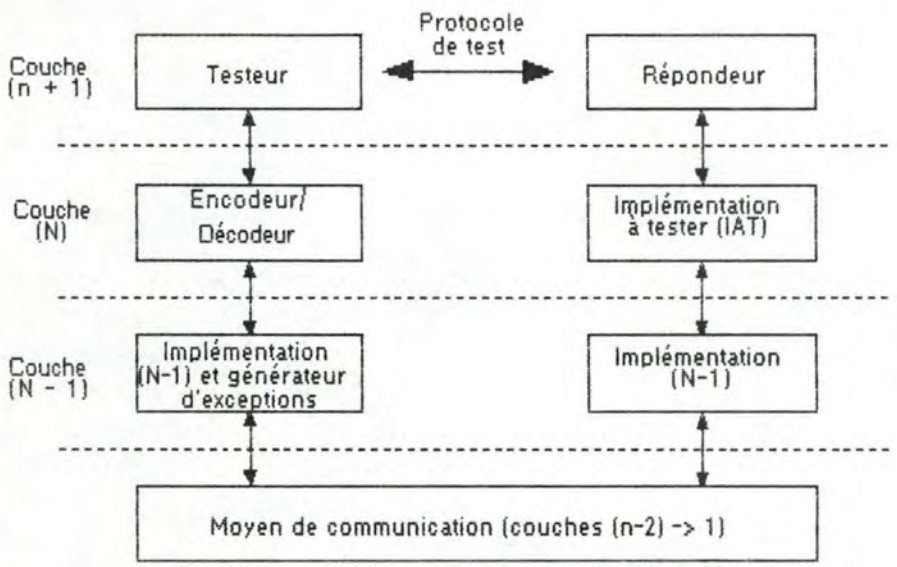


Figure 4.4

Architecture de test de conformité
proposée par le National Physical Laboratory (NPL)

On y distingue différents composants. Le testeur, dans le centre de test, interagit avec le répondeur disposé sur le site client, selon des modalités définies dans un protocole de test. Le testeur et le répondeur sont des programmes utilisant les services de la couche(N). Ils permettent de demander à l'implémentation à tester (IAT) de fournir certains services, de manière à pouvoir déterminer s'ils sont offerts correctement.

La couche(N) contient dans le centre de test, un encodeur/décodeur d'unités de donnée de protocole, et l'IAT sur le site client. L'encodeur/décodeur permet au testeur de construire et d'analyser les unités de donnée de protocole à émettre vers ou reçues de l'IAT. Le testeur peut ainsi étudier si la comportement de l'IAT est conforme au protocole prévu pour la couche(N), en émettant certaines unités de donnée de protocole vers cette implémentation, et en analysant les unités de donnée de protocole reçues de celle-ci.

La couche(N-1), dans le centre, de test est particulière. Elle contient un générateur d'exceptions, capable d'introduire des erreurs dans les unités de donnée de protocole émises par la couche(N). On peut ainsi vérifier si l'IAT répond de manière satisfaisante à des unités de donnée de protocole incorrects.

Le lecteur désirant de plus amples informations sur l'architecture NPL peut se référer à [4].

Le module de transcodage proposé ici peut réaliser les fonctionnalités de l'encodeur/décodeur de l'architecture NPL. Le testeur peut en effet demander d'émettre une unité de donnée de protocole, en fournissant au module la notation de valeur qui y correspond. De même, il pourra demander au module de visualiser les unités de donnée de protocole reçues de l'IAT, sous forme de notation de valeur. Le testeur peut également exploiter la structure de donnée offerte par le module de transcodage.

5. Réalisation de l'outil d'analyse et de production d'unités de données de protocole

5.1. Introduction

Nous présenterons ici la mise en oeuvre du module d'analyse et de production d'unités de donnée de protocole, ou module de transcodage, dont nous avons évoqué les fonctionnalités au chapitre 4.

Nous avons constaté que ce module est tributaire de la spécification des unités de donnée de protocole, rédigée à l'aide du langage X409. Cette partie est consacrée à la description de l'implémentation de ce module, dans le cadre d'une spécification particulière. Nous pourrons ainsi tirer un ensemble de conclusions intéressantes, permettant la généralisation du module à toute spécification rédigée dans le langage X409.

Le point 5.2 contient le texte de la spécification utilisée lors de la mise en oeuvre du module de transcodage.

Nous évoquerons au point 5.3 un ensemble de contraintes auxquelles l'implémentation doit satisfaire, de manière à permettre une généralisation aisée et une réutilisabilité maximale dans des contextes divers.

Nous présenterons ensuite la conception (point 5.4) et la réalisation (point 5.5) du module.

5.2. Spécification des unités de donnée de protocole

La spécification des unités de donnée de protocole que nous avons utilisée pour la réalisation du module de transcodage est tirée de la recommandation X409 [24]. Elle est relativement simple, mais intègre la majorité des concepts du langage X409 abordés au point 3.3.2.

Nous pourrons dégager les principes permettant de déduire la structure de donnée du texte de la spécification. D'autre part, nous pourrons mettre en évidence la marche à suivre pour

analyser ou produire une unité de donnée de protocole, et visualiser ou saisir une notation de valeur. Les constatations que nous ferons sur cet exemple simple nous permettrons de généraliser facilement le module de transcodage, étant donné qu'il suffira d'appliquer ces principes dans le cadre d'une spécification plus large.

L'exemple proposé dans la recommandation X409 définit les informations relatives aux membres du personnel d'une entreprise. On y trouve un certain nombre de types pré-définis (SET, SEQUENCE, INTEGER), un type considéré comme "standard" par la recommandation X409 (IA5String), et les deux formes de Tagged Type (implicite et explicite). La spécification est la suivante :

```
InformationAboutPersonnel DEFINITIONS ::=

BEGIN

 PersonnelRecord  ::= [APPLICATION 0] IMPLICIT SET {
                          Name,
                          title [0] IA5String,
                          EmployeeNumber,
                          dateOfHire [1] Date,
                          nameOfSpouse [2] Name OPTIONAL,
                          [3] IMPLICIT SEQUENCE OF
                           ChildInformation DEFAULT {}  }

 ChildInformation  ::=  SET {
                          Name,
                          dateOfBirth [0] Date }

 Name              ::= [APPLICATION 1] IMPLICIT SEQUENCE {
                          givenName IA5String,
                          initial IA5String,
                          familyName IA5String  }

 EmployeeNumber    ::= [APPLICATION 2] IMPLICIT INTEGER

 Date              ::= [APPLICATION 3] IMPLICIT IA5String
                          --          YYYYMMDD

END
```

Nous donnerons d'autres exemples, lorsque ceci sera nécessaire, pour mettre en évidence comment les fonctionnalités du module sont assurées pour des types autres que ceux définis dans cet exemple.

5.3. Contraintes à respecter lors de la mise en oeuvre

L'implémentation du module de transcodage doit respecter certaines contraintes, en vue de permettre une généralisation aisée et une réutilisabilité maximale du code dans des contextes fort divers. Ces contextes sont définis par la spécification des unités de donnée de protocole et par l'environnement logiciel et matériel dans lequel vient s'insérer le module.

5.3.1. Portabilité

La mise en oeuvre du module doit être portable, de manière à ce que le transfert de l'implémentation d'un environnement logiciel et matériel vers un autre ne nécessite qu'un effort minimal.

Cette contrainte est importante si l'on considère que le module est susceptible d'être utilisé dans des environnements fort divers (cf. point 4.4). En effet, les programmes d'application qui utilisent les services de ce module peuvent être implantés dans des systèmes a priori quelconques. Ce sera le cas dans un réseau de communication, où l'architecture OSI est justement utilisée pour permettre l'interconnexion d'ordinateurs de tous types. Ainsi, le module de transcodage, s'insérant dans la couche application, doit pouvoir s'intégrer facilement dans cet environnement.

5.3.2. Réutilisabilité

Le module de transcodage doit être conçu et implémenté de manière à être réutilisable dans différents contextes. On doit pouvoir réutiliser une portion significative du code lors d'une modification de la spécification sur laquelle se base l'implémentation.

Ainsi, seulement une partie restreinte du code doit faire l'objet d'une modification manuelle ou d'une génération automatique.

5.3.3. Modifiabilité

La mise en oeuvre du module de transcodage se base sur la version de la recommandation X409 publiée en 1984. Etant donné que celle-ci peut faire l'objet de modifications dans

les versions futures, il est important de veiller à la modifiabilité de l'implémentation.

Ainsi, le module doit pouvoir être facilement adapté aux nouvelles recommandations moyennant un effort minimal.

C'est la raison pour laquelle nous avons intégré dans cette implémentation la possibilité de convertir les représentations de valeurs de types de donnée définis de manière récursive. Bien que ceci ne soit pas explicitement autorisé (ni interdit) dans la recommandation X409, il nous semble intéressant d'offrir cette possibilité. En effet, le langage ASN.1 de l'ISO [17] [18] permet déjà les définitions récursives de types de donnée. On peut prévoir que dans sa version 1988, le langage X409 propose également cette possibilité.

5.4. Conception du module de transcodage

La conception du logiciel réalisant les fonctionnalités du module de transcodage, évoquées au point 4.3, passe par deux étapes principales. Il s'agit en effet de concevoir la structure de donnée mise à la disposition des programmes d'application, et l'architecture logicielle à mettre en place pour assurer les fonctionnalités évoquées.

Dans ce qui suit, nous exposerons ces deux étapes en précisant les fonctionnalités informelles présentées au point 4.3, tout en prenant soin de ne pas établir de lien explicite avec une implémentation particulière de celles-ci. Nous présenterons la mise en oeuvre au point 5.5.

5.4.1. Conception de la structure de donnée

La structure de donnée que le module fournit aux programmes d'application doit être conçue avec soin, étant donné que la qualité de celle-ci détermine directement son utilité pratique. Le module de transcodage n'est justifié que s'il offre aux programmes qui l'utilisent un certain confort dans la manipulation des unités de donnée de protocole.

On peut ainsi énoncer certaines qualités de la structure de donnée. Elle doit être concise, de manière à ne pas dépasser la taille du codage TLV. Il est nécessaire qu'elle soit homogène, pour qu'à chaque type pré-défini du langage X409 corresponde toujours la même structuration. Elle doit permettre un accès rapide à l'information qu'elle contient et offrir une possibilité de mise à jour efficace des valeurs qu'elle représente. Enfin, elle doit être générale

pour permettre de représenter les valeurs de n'importe quel type de donnée défini dans la spécification.

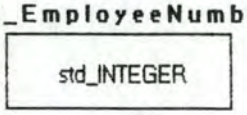
Nous ne pouvons pas énoncer les règles précises permettant de déduire la structure de donnée de la spécification des unités de donnée de protocole. En effet, celles-ci dépendent fortement des possibilités de structuration offertes par le langage de programmation utilisé lors de l'implémentation. Nous aborderons les règles précises d'établissement de la structure de donnée lorsque nous présenterons la mise en oeuvre au point 5.5.

Il est cependant utile de présenter quelques principes généraux intervenant dans l'établissement de cette structure. Nous désignerons dans la suite ces structures de donnée par le nom du type pré-défini, précédé de "std_" ("STructure de Donnée associée à").

Ainsi, la structure de donnée associée au type pré-défini BOOLEAN porte le nom std_BOOLEAN.

Les types de donnée pré-définis primitifs BOOLEAN, INTEGER, BIT STRING et OCTET STRING sont les briques de base du langage X409. Ils véhiculent de l'information utile, et c'est pourquoi on peut leur attribuer une représentation propre.

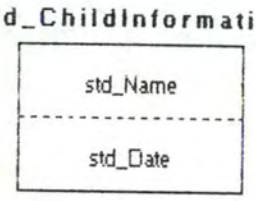
A chaque type de donnée défini dans la spécification des unités de donnée de protocole correspond une structure particulière qui est exactement la même que celle choisie pour le type de base.



La structure de donnée associée à un type de base constructeur n'est pas toujours la même. En effet, dans le cas de la SEQUENCE ou du SET, le type représente un ensemble, ordonné ou non, de valeurs dont les types sont ceux spécifiés. Le format de la structure de donnée dépend donc du nombre d'éléments, et surtout de leurs types. Un type de base constructeur est donc représenté par une strcuturation traduisant le regroupement de plusieurs éléments, plutôt que par une structure propre.

Pour plus de simplicité, il est intéressant d'imposer un ordre particulier pour les valeurs des membres d'un SET. Nous choisirons simplement l'ordre de définition des membres dans la spécification.

Nous représenterons la structure de donnée associée à ChildInformation de la manière suivante :



Dans le cas d'un type de base CHOICE, la structure de donnée doit pouvoir représenter une valeur d'un des types spécifiés dans la définition.

Nous choisirons pour représenter une valeur d'un type CHOICE, une structure de donnée contenant deux éléments. Le premier indique le type de la valeur, et le second contient la repésentation de la valeur elle-même.

La définition d'une unité de donnée du protocole P1 (MPDU) présentée au point 3.3 est la suivante :

MPDU ::= CHOICE {[0] IMPLICIT UserMPDU, ServiceMPDU}

La structure de donnée associée à MPDU porte le nom std_MPDU et peut être représentée de la manière suivante :

5.4.2. Conception de l'architecture logicielle

L'architecture à mettre en place pour réaliser les fonctionnalités du module de transcodage doit permettre la mise en oeuvre d'un logiciel respectant les contraintes évoquées au point 5.3.

La découpe choisie est modulaire. Elle permet d'isoler dans des modules spécifiques les fonctionnalités non portables, susceptibles d'être modifiées fréquemment ou réutilisables.

Ainsi, lors du transfert de l'implémentation d'un système de traitement vers un autre, il suffit de modifier les modules contenant des fonctions non portables.

Lors d'un changement de la spécification des unités de donnée de protocole, uniquement les modules intégrant des fonctions tributaires de la spécification sont à revoir. De plus, les fonctions liées directement aux propositions de la recommandation X409 sont regroupées dans un module spécifique. Si des modifications interviennent dans les versions futures de cette recommandation, il suffit simplement de mettre à jour ce module.

On peut également isoler un certain nombre de fonctions réutilisables quelle que soit la spécification d'unités de donnée de protocole. Les fonctions associées à la manipulation des types standards ou pré-définis proposés dans la recommandation X409 peuvent être conservées lors d'une modification du texte de la spécification.

La découpe modulaire a d'autres avantages. Les différents composants logiciels peuvent être mis au point indépendamment les uns des autres. Ainsi, un module peut faire abstraction d'un certain nombre de détails, en utilisant les services offerts par d'autres modules. De plus, chaque composant peut être testé sans tenir compte des autres modules. Ce n'est que lorsque le logiciel complet est intégré que l'on doit procéder à des tests incluant tous les modules.

D'autre part, une découpe modulaire permet d'isoler un certain nombre de sous-systèmes utiles, assurant chacun une fonctionnalité significative du logiciel complet. On peut d'abord se limiter à la mise en oeuvre d'une fonctionnalité importante, et mettre déjà cette partie du logiciel à la disposition des programmeurs d'application, et compléter l'outil par la suite.

L'architecture modulaire que nous proposons pour le logiciel de transcodage est représentée à la figure 5.1. On y distingue un certain nombre de modules, assumant chacun une fonctionnalité propre, en utilisant les services d'autres composants.

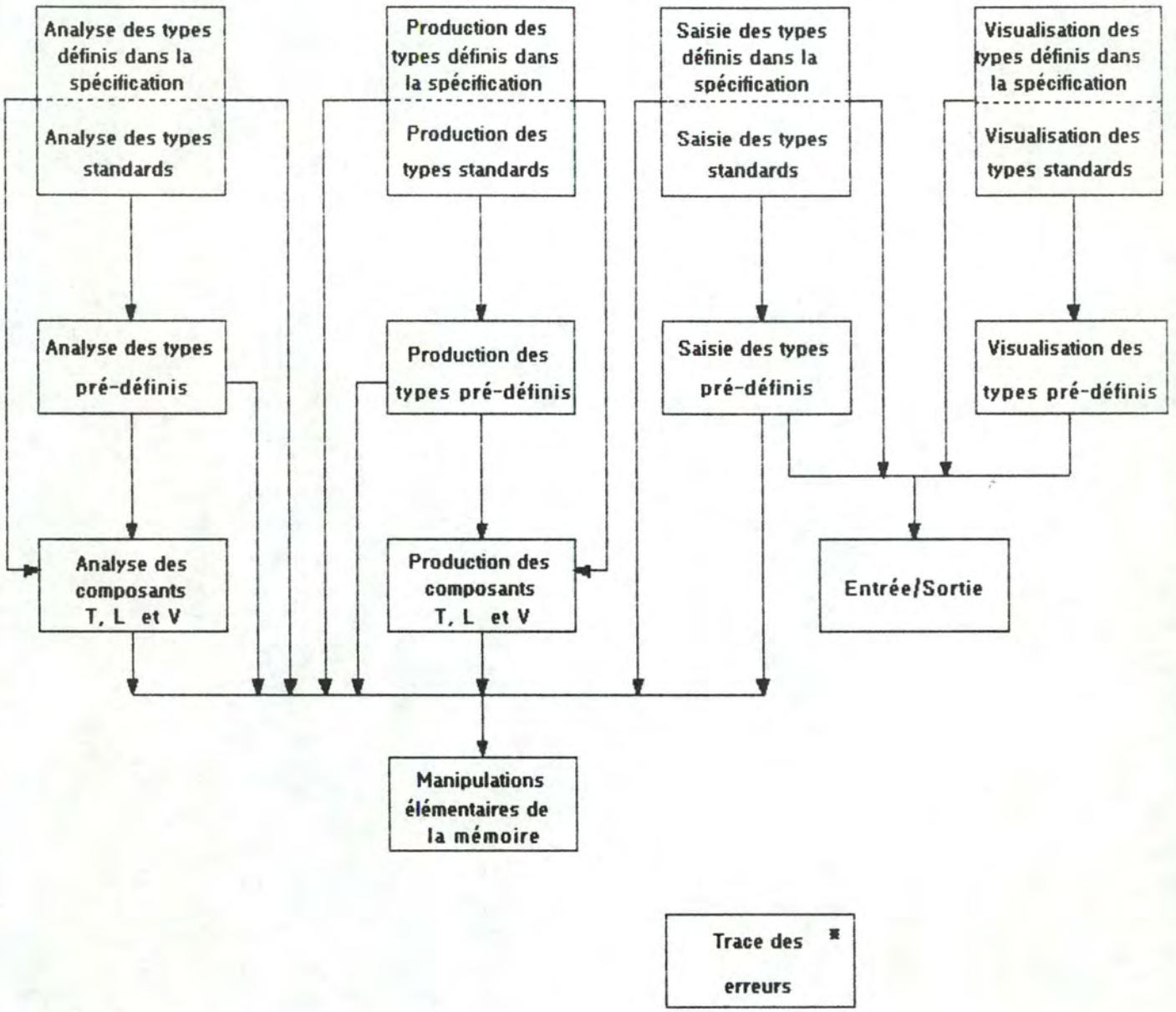


Figure 5.1

Architecture logicielle

Nous allons présenter dans ce qui suit les fonctionnalités réalisées par chacun des module représentés. Il est à noter que tous les composants utilisent les services du module "Trace des erreurs". Pour plus de clarté, nous n'avons pas représenté ces relations.

5.4.2.1. Module d'analyse des types définis dans la spécification et des types standards

Le module d'analyse des types définis dans la spécification et des types standards offre à ses utilisateurs des primitives permettant d'analyser un ensemble d'octets représenatnt le codage TLV d'une valeur d'un de ces types.

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour analyser une unité de donnée de protocole et obtenir l'information qu'elle contient.

On peut considérer que les définitions de types standards proposés dans la recommandation X409 viennent simplement s'ajouter à la spécification des unités de donnée de protocole. En effet, leur spécification est proposée une fois pour toutes, et peut être considérée comme "globale" à n'importe quelle autre spécification.

On peut ainsi regrouper les primitives associées aux types standards et celles manipulant les types définis dans un même module, étant donné que les moyens mis en oeuvre pour analyser les valeurs de ces types sont similaires.

Les primitives d'analyse associées aux types standards sont cependant réutilisables. C'est pourquoi sur la figure 5.1, le module d'analyse des types définis dans la spécification et des types standards est scindé en deux parties. La première doit être modifiée lors d'un changement dans la spécification des unités de donnée de protocole, la seconde peut être conservée.

Le module contient donc un ensemble de primitives d'analyse. A chaque type standard ou défini dans la spécification correspond exactement une primitive. Nous présenterons la spécification d'une d'entre elles sur un exemple tiré de la spécification du point 5.2. Il s'agit de la définition du nom d'une personne (Name), constituée du prénom, des initiales et du nom de famille.

La primitive qui permet l'analyse d'un élément de donnée représentant une valeur de ce type s'intitule analyser_Name et sa forme générale est la suivante :

analyser_Name (optionnel,début,fin,tagSupplémentaire,
classeTag,identificateurTag,
nouveauDébut,variableRésultat)

Entrée

- optionnel (oui/non)

Ce paramètre indique si la primitive est invoquée dans le cadre de l'analyse de la valeur d'un élément optionnel, ou dans un autre contexte.

- début,fin (numéros d'ordre d'octet)

Ces paramètres délimitent l'ensemble d'octets à analyser. On le désignera dans la suite par TLV [deb .. fin].

- tagSupplémentaire (absent/implicite/explicite)

Ce paramètre indique si la primitive doit tenir compte d'un tag supplémentaire lors de l'analyse, et indique la forme que celui-ci prend (implicite ou explicite).

- classeTag (UNIVERSAL,APPLICATION,PRIVATE,CONTEXT)

Lorsque la primitive doit tenir compte d'un tag supplémentaire, ce paramètre indique la classe du tag.

- identificateurTag (valeur numérique)

Lorsque la primitive doit tenir compte d'un tag supplémentaire, ce paramètre indique l'identificateur du tag.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

Ce paramètre indique le sous-ensemble conteant effectivement la valeur du type Name. Il s'agit de TLV [début .. nouveauDébut - 1].

- variableRésultat (de type std_Name)

Ce paramètre donne la référence de la structure de donnée qui doit accueillir le transcodage de l'information représentée dans TLV [début ... nouveauDébut - 1].

Retour

La primitive, après l'analyse, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'il existe un sous-ensemble de TLV [début .. fin] dont l'origine est début, et contenant la représentation correcte d'une valeur du type Name.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est d'analyser l'ensemble d'octets TLV [début .. fin]. Elle détermine s'il existe un sous-ensemble dont l'origine est début et qui contient la représentation correcte d'une valeur du type Name.

Si c'est le cas, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise les paramètres résulat nouveauDébut et variableRésultat. Le sous-ensemble TLV [début .. nouveauDébut - 1] est maximal, en ce sens qu'il n'existe pas de sous-ensemble plus grand que lui, représentant une valeur correcte du type Name. La structure de donnée désignée par variableRésultat contient l'information codée dans ce sous-ensemble, représentée conformément à la définition de la structure de donnée std_Name.

Lorsqu'un tel sous-ensemble n'existe pas, ou quand les paramètres sont incorrects, la primitive retourne ERREUR, après avoir justifié sa réponse par un message d'eereur pertinent. Celui-ci n'est pas produit dans le cas où le paramètre optionnel indique que la primitive a été invoquée dans le cadre de l'analyse d'une valeur d'un type optionnel.

La spécification de la primitive a été rédigée pour l'analyse d'une valeur du type Name. Elle est identique pour tous les autres types définis dans la spécification des unités de donnée de protocole et les types standards proposés dans la recommandation X409. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée résultat. Ainsi, la primitive permettant l'analyse d'une valeur du type PersonnelRecord s'intitule analyser_PersonnelRecord, et le paramètre variableRésultat est de type std_PersonnelRecord.

Il est clair que dans les deux cas, l'analyse se fera différemment, étant donné que le type de base de Name est une SEQUENCE, et que le type de base de PersonnelRecord est un SET. Nous avons vu au point 3.3.2.1 que l'ordre d'apparition des éléments d'une SEQUENCE est significatif, tandis que pour le SET, l'ordre dans lequel sont codées les valeurs des membres n'a aucune importance.

Notons également que la primitive analyser_name peut être utilisée dans des contextes forts divers. Tout d'abord, un programme d'application peut demander l'analyse d'une unité de donnée de protocole du type Name en invoquant la primitive de la manière suivante :

```
analyser_Name (non-optionnel,début,fin,pas de tag,/,/,
                        nouveauDébut,variableRésultat)
```

En effet, on ne doit pas tenir compte d'un tag supplémentaire lors de l'analyse.

D'autres primitives d'analyse peuvent également utiliser les services de analyser_Name. C'est le cas lors de l'analyse d'une valeur du type PersonnelRecord. La primitive analyser_PersonnelRecord doit en effet étudier succesivement les valeurs des membres du SET. Etant donné que pour chaque type de membre, une primitive est disponible, analyser_PersonnelRecord peut invoquer celles-ci. Elle doit cependant fournir des informations complémentaires concernant les tags des membres. Si elle désire analyser l'ensemble d'octets [k .. fin], supposé contenir une valeur valeur du premier membre, qui est non optionnel et ne possède pas de tag supplémentaire, elle invoque la primitive avec les paramètres suivants :

```
analyser_Name (non-optionnel,k,fin,absent,/,/,
                        nouveauDébut,variableRésultat)
```

Lors de l'analyse du membre portant le nom de référence nameOfSpouse, qui est optionnel et qui possède un tag explicite d'identificateur 2 de la classe CONTEXT, elle invoque la primitive de la manière suivante :

```
analyser_Name (optionnel,k,fin,explicite,CONTEXT,2,
                        nouveauDébut,variableRésultat)
```

5.4.2.2. Module d'analyse des types pré-définis

Le module d'analyse des types pré-définis offre à ses utilisateurs un ensemble de primitives permettant l'analyse d'un ensemble d'octets supposés représenter une valeur d'un de ces types.

Les utilisateurs potentiels de ce modules sont d'une part les programmes d'application, désireux d'analyser le codage TLV d'une valeur d'un des types pré-définis, et d'autre part le module d'analyse des types définis dans la spécification et des types standards.

A chaque type pré-défini primitif correspond une primitive d'analyse dans le module. Celles-ci assument les mêmes fonctionnalités et possèdent les mêmes interfaces que les primitives évoquées au point 5.4.2.1.

La primitive associée à l'analyse d'une valeur du type INTEGER prend la forme suivante :

```
analyser_INTEGER (optionnel,début,fin,tagSupplémentaire,
                  classeTag,identificateurTag,
                  nouveauDébut,variableRésultat)
```

Une telle primitive est disponible pour chacun des types pré-définis primitifs (INTEGER, BOOLEAN, BIT STRING, OCTET STRING).

Il faut noter que pour le type NULL, l'interface de la primitive est différent, étant donné qu'aucune structure de donnée n'est associée à la représentation d'une valeur de ce type.

Ainsi, la primitive prend la forme suivante :

```
analyser_NULL (optionnel,début,fin,tagSupplémentaire,
               classeTag,identificateurTag,
               nouveauDébut)
```

Elle signale simplement la présence du codage TLV de la valeur NULL, sans toutefois fournir le transcodage.

Le module ne contient pas de primitives associées aux types de base constructeurs, étant donné que ceux-ci n'ont pas de représentation propre. L'analyse à effectuer dépend des types et du nombre d'éléments spécifiés.

5.4.2.3. Module d'analyse des composants T, L et V

Le module d'analyse des composants T, L et V offre à ses utilisateurs des primitives permettant l'analyse de ces composants au sein d'un ensemble d'octets.

Analyse du composant T

Une primitive est disponible pour analyser le composant Type. Elle prend la forme suivante :

analyser_Composant_T (optionnel,début,fin,classe,
forme,identificateur,
nouveauDébut)

Entrée

- optionnel (oui/non)

  Ce paramètre indique si la primitive est invoquée dans le cadre de l'analyse de la valeur d'un élément optionnel, ou dans un autre contexte.

- début,fin (numéros d'ordre d'octet)

  Ces paramètres délimitent l'ensemble d'octets à analyser. On le désignera dans la suite par TLV [deb .. fin].

- classe (UNIVERSAL,APPLICATION,PRIVATE,CONTEXT)

  Ces paramètres indiquent la valeur supposée des bits cc du composant Type.

- forme (PRIMITIVE/CONSTRUCTOR)

  Ce paramètre indique la valeur supposée du bit f du composant Type.

- identificateur (valeur numérique)

  Ce paramètre indique la valeur supposée des bits i du composant Type.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

  Ce paramètre indique le sous-ensemble contenant effectivement le composant Type. Il s'agit de TLV [début .. nouveauDébut - 1].

Retour

La primitive, après l'analyse, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

  La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'il existe un sous-ensemble de TLV [début .. fin] dont l'origine est début, et contenant la représentation correcte d'un composant Type dont les bits cc, f et i contiennent respectivement les valeurs classe, forme et identificateur.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est d'analyser l'ensemble d'octets TLV [début .. fin]. Elle détermine s'il existe un sous-ensemble dont l'origine est début et qui est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Type. Les bits cc, f et i doivent respectivement prendre les valeurs classe, forme et identificateur.

Si c'est le cas, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise nouveauDébut. Le sous-ensemble contenant le composant Type est donc TLV [début .. nouveauDébut - 1].

Lorsqu'un tel sous-ensemble n'existe pas, ou quand les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent. Celui-ci n'est cependant pas produit lorsque le paramètre optionnel indique l'utilisation de la primitive dans le cadre de l'analyse d'une valeur d'un élément optionnel.

Analyse du composant L

Une primitive est disponible pour analyser le composant Longueur. Elle prend la forme suivante :

analyser_Composant_L (optionnel,début,fin,
nouveauDébut,longueur_codée)

Entrée

- optionnel (oui/non)

Ce paramètre indique si la primitive est invoquée dans le cadre de l'analyse de la valeur d'un élément optionnel, ou dans un autre contexte.

- début,fin (numéros d'ordre d'octet)

Ces paramètres délimitent l'ensemble d'octets à analyser. On le désignera dans la suite par TLV [deb .. fin].

Sortie

- nouveauDébut (numéro d'ordre d'octet)

Ce paramètre indique le sous-ensemble contenant effectivement le composant Type. Il s'agit de TLV [début .. nouveauDébut - 1].

- longueur codée (valeur numérique ou INDEFINIE)

Ce paramètre désigne la variable destinée à accueillir la longueur extraite du composant L à analyser.

Retour

La primitive, après l'analyse, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'il existe un sous-ensemble de TLV [début .. fin] dont l'origine est début, et contenant la représentation correcte d'un composant Longueur.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est d'analyser l'ensemble d'octets TLV [début .. fin]. Elle détermine s'il existe un sous-ensemble dont l'origine est début et qui est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Longueur.

Si c'est le cas, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise nouveauDébut et longueur_codée. Le sous-ensemble content le composant L est donc TLV [début .. nouveauDébut - 1] et longueur_codée contient la longueur qui y est représentée. Dans le cas où le composant L est le codage d'une longueur de forme indéfinie, la variable longueur_codée est intialisée à la valeur conventionnelle INDEFINIE.

Dans le cas où un tel sous-ensemble n'existe pas, ou lorsque les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent. Celui-ci n'est cependant pas produit lorsque le paramètre optionnel indique l'utilisation de la primitive dans le cadre de l'analyse d'une valeur d'un élément optionnel.

Analyse du composant V

Le module fournit également des primitives permettant l'analyse d'un composant Valeur dans un ensemble d'octets. Elles extraient l'information qui y est codée et la transfèrent dans la structure de donnée.

Elles ne sont cependant applicables que dans le cas de composants V élémentaires, étant donné qu'eux seuls contiennent de l'information "utile".

Une primitive existe pour les types pré-définis primitifs BOOLEAN, INTEGER, BIT STRING et OCTET STRING. D'autres primitives ne sont pas nécessaires étant donné que les types de donnée de la spécification sont définis en fonction de ces types pré-définis. Les composants V élémentaires contiennent donc toujours de l'information pouvant être interprétée conformément aux types pré-définis.

Il n'est pas non plus nécessaire de disposer d'une primitive permettant l'analyse du composant V d'une valeur du type NULL, étant donné que celui-ci est toujours absent.

La primitive permettant l'analyse d'un composant V d'un type INTEGER prend la forme suivante :

analyser_Composant_V_INTEGER (optionnel,début,fin,
variableRésultat)

Entrée

- optionnel (oui/non)

  Ce paramètre indique si la primitive est invoquée dans le cadre de l'analyse de la valeur d'un élément optionnel, ou dans un autre contexte.

- début,fin (numéros d'ordre d'octet)

  Ces paramètres délimitent l'ensemble d'octets à analyser. On le désignera dans la suite par TLV [deb .. fin].

Sortie

- variableRésultat (de type std_INTEGER)

Ce paramètre désigne la variable destinée à accueillir l'information extraite du composant V à analyser.

Retour

La primitive, après l'analyse, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR l'ensemble TLV [début .. fin] contient la représentation correcte d'une valeur du type INTEGER.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est d'analyser l'ensemble d'octets TLV [début .. fin]. Elle détermine si l'ensemble est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Valeur du type INTEGER.

Si c'est le cas, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise variableRésultat avec la valeur codée dans le composant V analysé.

Dans le cas où un tel sous-ensemble n'existe pas, ou lorsque les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent. Celui-ci n'est cependant pas produit lorsque le paramètre optionnel indique l'utilisation de la primitive dans le cadre de l'analyse d'une valeur d'un élément optionnel.

La même primitive est disponible pour les types BOOLEAN, BIT STRING et OCTET STRING. Leur spécification est identique, si ce n'est que l'intitulé de la primitive et le type de la structure de donnée résultat changent.

5.4.2.4. Module de production des types définis dans la spécification et des types standards

Le module de production des types définis dans la spécification et des types standards offre à ses utilisateurs des primitives permettant de produire un ensemble d'octets représentant le codage TLV d'une valeur d'un de ces types.

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour produire des unités de donnée de protocole à partir d'une valeur de la structure de donnée.

Les remarques faites au point 5.4.2.1. concernant les types standards sont également applicables ici.

Nous spécifierons ici la primitive permettant de produire le codage TLV d'une valeur du type Name, à partir d'une valeur de la structure de donnée.

Il est à noter que pour des raisons pratiques, l'ensemble d'octets est produit en ordre inverse. Ainsi, une fois le composant V d'un élément de donnée codé, on peut aisément déterminer la longueur à coder dans le composant L.

Si l'on produit les octets dans le bon ordre, on ne peut pas déterminer, lors du codage du composant L, la longueur à y représenter, étant donné que le composant V n'est pas encore produit.

Ainsi, un programme d'application invoquant une primitive de production, doit relire l'ensemble d'octets résultant en ordre inverse, de manière à obtenir l'unité de donnée de protocole correct.

La forme générale de la primitive de production associée au type Name est la suivante :

```
produire_Name (valeurACoder,début,tagSupplémentaire,
                        classeTag,identificateurTag,
                        nouveauDébut)
```

Entrée

- valeurACoder (du type std_name)

  Ce paramètre donne la référence de la structure de donnée qui contient l'information à coder dans l'ensemble d'octets.

- début (numéro d'ordre d'octet)

  Ce paramètre indique le premier octet à partir duquel la primitive peut produire le codage TLV.

- tagSupplémentaire (absent/implicite/explicite)

  Ce paramètre indique si la primitive doit tenir compte d'un tag supplémentaire lors de la production, et indique la forme que celui-ci prend (implicite ou explicite).

- classeTag (UNIVERSAL,APPLICATION,PRIVATE,CONTEXT)

  Lorsque la primitive doit tenir compte d'un tag supplémentaire, ce paramètre indique la classe du tag.

- identificateurTag (valeur numérique)

  Lorsque la primitive doit tenir compte d'un tag supplémentaire, ce paramètre indique l'identificateur du tag.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

  Ce paramètre indique le sous-ensemble contenant le codage produit. Il s'agit de TLV [début .. nouveauDébut - 1].

Retour

La primitive, après la production, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

  La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'elle a produit l'ensemble TLV [début .. nouveauDébut - 1] contenant la représentation correcte de la structure de donnée désignée par valeurACoder.

- ERREUR

  La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de produire l'ensemble d'octets correspondant au codage TLV de la valeur du type Name,contenue dans la structure de donnée désignée par valeurACoder. Le résultat est l'ensemble TLV [début .. nouveauDébut - 1].

Cet ensemble doit être relu en sens inverse pour obtenir le codage effectif.

La spécification est identique pour les autres types définis dans la spécification des unités de donnée de protocole et les types standards. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée contenant la valeur à coder.

5.4.2.5. Module de production des types pré-définis

Le module de production des types pré-définis offre à ses utilisateurs un ensemble de primitives permettant de produire un ensemble d'octets à partir d'une valeur de la structure de donnée.

Les utilisateurs potentiels de ce module sont d'une part les programmes d'application, désireux de produire le codage TLV d'une valeur d'un des types pré-définis, et d'autre part le module de production des types définis dans la spécification et des types standards.

A chaque type pré-défini primitif correspond une primitive de production dans le module. Celles-ci assument les mêmes fonctionnalités et possèdent les mêmes interfaces que les primitives évoquées au point 5.4.2.4.

La primitive associée à la production d'une valeur du type INTEGER prend la forme suivante :

```
produire_INTEGER (valeurACoder,début,tagSupplémentaire,
                  classeTag,identificateurTag,
                  nouveauDébut)
```

Une telle primitive est disponible pour chacun des types pré-définis primitifs (INTEGER, BOOLEAN, BIT STRING, OCTET STRING).

Il faut noter que pour le type NULL, l'interface de la primitive est différent, étant donné qu'aucune structure de donnée n'est associée à la représentation d'une valeur de ce type.

Ainsi, la primitive prend la forme suivante :

```
produire_NULL (début,tagSupplémentaire,
                        classeTag,identificateurTag,
                        nouveauDébut)
```

Elle produit simplement le codage TLV de la valeur NULL.

Le module ne contient pas de primitives associées aux types de base constructeurs, étant donné que ceux-ci n'ont pas de représentation propre. La production à effectuer dépend des types et du nombre d'éléments spécifiés.

5.4.2.6. Module de production des composants T, L et V

Le module de production des composants T, L et V offre à ses utilisateurs des primitives permettant de produire un ensemble d'octets représentant le codage de ces composants.

Production du composant T

Une primitive est disponible pour produire le codage du composant Type. Elle prend la forme suivante :

```
produire_Composant_T (début,classe,
                                forme,identificateur,
                                nouveauDébut)
```

Entrée

- début (numéro d'ordre d'octet)

  Ce paramètre indique le premier octet à partir duquel le composant T peut être produit.

- classe (UNIVERSAL,APPLICATION,PRIVATE,CONTEXT)

  Ces paramètres indiquent la valeur des bits cc à coder dans le composant Type.

- forme (PRIMITIVE/CONSTRUCTOR)

  Ce paramètre indique la valeur du bit f à coder dans le composant Type.

- identificateur (valeur numérique)

Ce paramètre indique la valeur des bits i à coder dans le composant Type.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

Ce paramètre indique le sous-ensemble contenant effectivement le codage du composant Type. Il s'agit de TLV [début .. nouveauDébut - 1].

Retour

La primitive, après la production, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsque le codage s'est effectué correctement.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de produire un ensemble d'octets qui est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Type. Les bits cc, f et i valent respectivement classe, forme et identificateur.

Si le composant est produit, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise la variable nouveauDébut. Le sous-ensemble contenant le codage du composant T est donc TLV [début .. nouveauDébut - 1].

Dans les autres cas, ou lorsque les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent.

Production du composant L

Une primitive est disponible pour produire le codage du composant Longueur. Elle prend la forme suivante :

```
produire_Composant_L (longueurACoder,début,
                                    nouveauDébut)
```

Entrée

- longueurACoder (valeur numérique ou INDEFINIE)

  Ce paramètre indique la longueur à coder dans le composant L. Lorsqu'il prend la valeur conventionnelle INDEFINIE, on doit coder une longueur de forme indéfinie.

- début (numéro d'ordre d'octet)

  Ce paramètres indique le premier octet à partir duquel le composant L peut être produit.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

  Ce paramètre indique le sous-ensemble contenant effectivement le codage du composant Type. Il s'agit de TLV [début .. nouveauDébut - 1].

Retour

La primitive, après l'analyse, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

  La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsque le codage du composant Longueur s'est effectuée correctement.

- ERREUR

  La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de produire un ensemble d'octets qui est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Longueur.

Si le composant est produit, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise nouveauDébut. Le sous-ensemble contenant le composant L est donc TLV [début .. nouveauDébut - 1].

Dans les autres cas, ou lorsque les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent.

Production du composant V

Le module fournit également des primitives permettant de produire le codage d'un composant Valeur. Elles y représentent la valeur fournie dans la structure de donnée.

Elles ne sont cependant applicables que dans le cas de composants V élémentaires, étant donné qu'eux seuls peuvent contenir de l'information "utile".

Une primitive existe pour les types pré-définis primitifs BOOLEAN, INTEGER, BIT STRING et OCTET STRING. D'autres primitives ne sont pas nécessaires étant donné que les types de donnée de la spécification sont définis en fonction de ces types pré-définis. Les composants V élémentaires contiennent donc toujours de l'information pouvant être interprétée conformément aux types pré-définis.

Il n'est pas non plus nécessaire de disposer d'une primitive permettant de produire le composant V d'une valeur du type NULL, étant donné que celui-ci est toujours absent.

La primitive permettant la production d'un composant V pour un type INTEGER prend la forme suivante :

produire_Composant_V_INTEGER (valeurACoder,début,
nouveauDébut)

Entrée

- valeurACoder (de type std_INTEGER)

  Ce paramètre désigne la structure de donnée contenant l'information à coder dans le composant V.

- début (numéro d'ordre d'octet)

  Ce paramètre indique le premier octet à partir duquel le composant V peut être produit.

Sortie

- nouveauDébut (numéro d'ordre d'octet)

  Ce paramètre indique le sous-ensemble contenant après la production, le codage du composant V.

Retour

La primitive, après la production, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsque le codage du composant V s'est effectué correctement.

- ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de produire un ensemble d'octets qui est le codage conforme aux règles énoncées au point 3.2.1, d'un composant Valeur du type INTEGER.

Si elle peut le produire, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise nouveauDébut. Ainsi, le sous-ensemble contenant effectivement le codage du composant V est TLV [début .. nouveauDébut - 1].

Dans les autres cas, ou lorsque les paramètres sont incorrects, la primitive retourne la valeur conventionnelle ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent.

La même primitive est disponible pour les types BOOLEAN, BIT STRING et OCTET STRING. Leur spécification est identique, si ce n'est que l'intitulé de la primitive et le type de la structure de donnée contenant la valeur à coder changent.

5.4.2.7. Module de saisie des types définis dans la spécification et des types standards

Le module de saisie des types définis dans la spécification et des types standards offre à ses utilisateurs des primitives permettant la saisie d'une notation de valeur d'un de ces types et la représentation de l'information qu'elle contient dans la structure de donnée .

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour saisir une notation de valeur et obtenir l'information qu'elle contient.

La saisie s'effectue sur un quelconque support externe (terminal, fichier, ...). Le module de saisie utilise les services du module d'entrée/sortie pour obtenir l'information nécessaire. C'est ce dernier qui choisit le support externe d'où proviendra l'information.

Pour les raisons déjà signalées au point 5.4.2.1, le module intègre également des primitives de saisie des types standards. On peut ici aussi considérer qu'ils viennent compléter la spécification des unités de donnée de protocole.

A chaque type standard ou défini dans la spécification correspond exactement une primitive de saisie. Nous ne présenterons ici que la primitive associée au type Name, défini dans la spécification donnée au point 5.2. Elle prend la forme générale suivante :

saisir_Name (optionnel,variableRésultat)

Entrée

- optionnel (oui/non)

  Ce paramètre indique si la primitive est invoquée dans le cadre de la saisie de la valeur d'un élément optionnel, ou dans un autre contexte.

Sortie

- variableRésultat (du type std_Name)

  Ce paramètre indique la référence de la structure de donnée qui doit accueillir le transcodage de la notation de valeur saisie.

Retour

La primitive, après la saisie, retourne une valeur indiquant son résultat :

- PAS D'ERREUR

  La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'une notation de valeur correcte du type Name a été saisie.

- ERREUR

  La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de saisir une notation de valeur du type Name.

Si la notation saisie est correcte, elle retourne la valeur conventionnelle PAS D'ERREUR et initialise variableRésultat. La structure de donnée désignée par variableRésultat contient alors l'information exprimée dans la notation saisie, représentée conformément à la définition de la structure de donnée std_Name.

Lorsque la notation saisie n'est pas une notation correcte d'une valeur du type Name, ou quand les paramètres sont incorrects, la primitive retourne ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent. Celui-ci n'est pas produit dans le cas où le paramètre optionnel indique que la primitive a été invoquée dans le cadre de la saisie d'une notation de valeur d'un type optionnel.

La spécification de la primitive a été rédigée pour la saisie d'une notation de valeur du type Name. Elle est identique pour tous les autres types définis dans la spécification des unités de donnée de protocole et les types standards proposés dans la recommandation X409. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée résultat. Ainsi, la primitive permettant l'analyse d'une valeur du type PersonnelRecord s'intitule saisir_PersonnelRecord, et le paramètre variableRésultat est de type std_PersonnelRecord.

5.4.2.8. Module de saisie des types pré-définis

Le module de saisie des types pré-définis offre à ses utilisateurs des primitives permettant la saisie d'une notation de valeur d'un de ces types et la représentation de l'information qu'elle contient.

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour saisir une notation de valeur d'un des types pré-définis et obtenir l'information qu'elle contient.

A chaque type pré-défini primitif correspond exactement une primitive de saisie. Celles-ci assument les mêmes fonctionnalités et possèdent le même interface que les primitives évoquées au point 5.4.2.7. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée résultat.

La primitive associée à la saisie d'une notation de valeur du type INTEGER prend la forme suivante :

saisir_INTEGER (optionnel,variableRésultat)

Une telle primitive est disponible pour chacun des types pré-définis BOOLEAN, INTEGER, BIT STRING et OCTET STRING. Signalons que l'interface de la primitive associée au type NULL est différent :

saisir_NULL (optionnel)

Ceci se justifie étant donné qu'aucune structure de donnée n'est associée au type NULL. La primitive signale simplement la saisie d'une notation de valeur du type NULL, sans donner la représentation dans une structure de donnée.

5.4.2.9. Module de visualisation des types définis dans la spécification et des types standards

Le module de visualisation des types définis dans la spécification et des types standards offre à ses utilisateurs des primitives permettant la visualisation d'une notation de valeur d'un de ces types à partir d'une valeur de la structure de donnée.

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour visualiser une notation de valeur.

La visualisation s'effectue sur un quelconque support externe (terminal, fichier, ...). Le module de visualisation utilise les services du module d'entrée/sortie pour afficher l'information. C'est ce dernier qui choisit le support externe où sera visualisée l'information.

Pour les raisons déjà signalées au point 5.4.2.1, le module intègre également des primitives de visualisation des types standards. On peut ici aussi considérer qu'ils viennent compléter la spécification des unités de donnée de protocole.

A chaque type standard ou défini dans la spécification correspond exactement une primitive de visualisation. Nous ne présenterons ici que la primitive associée au type Name, défini dans la spécification donnée au point 5.2. Elle prend la forme générale suivante :

visualiser_Name (valeurAVisualiser)

Entrée

-valeurAVisualiser (du type std_Name)

Ce paramètre donne la valeur de la structure de donnée à visualiser.

Retour

La primitive, après la visualisation, retourne une valeur indiquant son résultat :

-PAS D'ERREUR

La primitive retourne la valeur conventionnelle PAS D'ERREUR lorsqu'une notation de valeur correspondant à valeurAVisualiser a été affichée correctement.

-ERREUR

La primitive retourne la valeur conventionnelle ERREUR dans tous les autres cas.

Spécification

L'objectif de la primitive est de visualiser une notation de valeur du type Name.

Si la notation est visualisée correctement, elle retourne PAS D'ERREUR. La structure de donnée désignée par valeurAVisualiser contient l'information à visualiser sous forme de notation de valeur.

Dans les autres cas, ou quand les paramètres sont incorrects, la primitive retourne ERREUR, après avoir justifié sa réponse par un message d'erreur pertinent.

La spécification de la primitive a été rédigée pour la visualisation d'une notation de valeur du type Name. Elle est identique pour tous les autres types définis dans la spécification des unités de donnée de protocole et les types standards proposés dans la recommandation X409. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée résultat. Ainsi, la primitive permettant de visualiser la notation d'une valeur du type PersonnelRecord s'intitule visualiser_PersonnelRecord, et le paramètre valeurAVisualiser est de type std_PersonnelRecord.

5.4.2.10. Module de visualisation des types pré-définis

Le module de visualisation des types pré-définis offre à ses utilisateurs des primitives permettant la visualisation d'une notation de valeur d'un de ces types à partir d'une valeur de la structure de donnée.

Les utilisateurs potentiels de ce module sont les programmes d'application invoquant le logiciel de transcodage pour visualiser une notation de valeur d'un des types pré-définis et obtenir l'information qu'elle contient.

A chaque type pré-défini primitif correspond exactement une primitive de visualisation. Celles-ci assument les mêmes fonctionnalités et possèdent le même interface que les primitives évoquées au point 5.4.2.9. Il suffit de modifier l'intitulé de la primitive et le type de la structure de donnée dont il faut visualiser la valeur.

La primitive associée à la visualisation d'une notation de valeur du type INTEGER prend la forme suivante :

visualiser_INTEGER (valeurAVisualiser)

Une telle primitive est disponible pour chacun des types pré-définis BOOLEAN, INTEGER, BIT STRING et OCTET STRING. Signalons que l'interface de la primitive associée au type NULL est différent :

visualiser_NULL ()

Ceci se justifie étant donné qu'aucune structure de donnée n'est associée au type NULL. La primitive visualise simplement la notation de valeur NULL du type NULL.

5.4.2.11. Module d'entrée/sortie

Le module d'entrée/sortie offre à ses utilisateurs un ensemble de primitives permettant l'échange d'information avec un support externe (terminal, fichier, ...).

Le module contient deux primitives pour lesquelles nous nous contenterons de donner l'interface :

sortie (chaîne_de_caractères)

entrée (chaîne_de_caractères)

Elles permettent d'émettre une chaîne de caractères vers un support externe, ou d'en recevoir une de ce support.

Ce sont ces primitives qui établissent un lien explicite avec un support externe particulier. Lorsque les entrées/sorties sont destinées à un autre support, les primitives doivent être modifiées en conséquence. Les utilisateurs de celles-ci ne sont pas concernés par ce changement.

5.4.2.12. Module de manipulation de la mémoire

Le module de manipulation de la mémoire contient un ensemble de primitives permettant l'accès direct à la mémoire centrale.

Nous ne détaillerons pas ces primitives. Notons seulement que ce module permet de transposer les numéros d'ordre d'octets utilisés dans les autres modules, en adresses réelles en mémoire centrale. Etant donné que la longueur des ensembles d'octets manipulés par celles-ci n'est pas connue à l'avance, le module doit permettre de gérer des ensembles dépassant la taille de la mémoire. Il peut éventuellement stocker ces informations sur un support de mémoire auxiliaire, mais se charge de rendre disponibles en mémoire centrale les octets auxquels les autres modules demandent l'accès.

Les primitives classiques d'allocation et de libération de portions de mémoire sont également disponibles. Le module permet d'autre part d'extraire la valeur de certains bits dans un octet ou d'y déposer une valeur.

5.4.2.13. Module de trace des erreurs

Le module de trace des erreurs regroupe un ensemble de primitives permettant d'afficher des messages d'erreur. Nous ne détaillerons pas ici ces primitives.

5.5. Réalisation du module de transcodage

Nous présenterons dans ce qui suit l'implémentation de la structure de donnée et de l'architecture logicielle présentées

au point 5.4. Cette mise en oeuvre doit respecter les contraintes évoquées au point 5.3.

Nous aborderons au point 5.5.1 l'implémentation de la structure de donnée dont disposent les programmes d'application utilisant le module de transcodage. Nous montrerons comment déduire une structure de donnée de la spécification des unités de donnée de protocole.

Le point 5.5.2 est consacré à la présentation de l'implémentation de l'architecture logicielle.

Nous nous sommes limités à la mise en oeuvre d'un sous-système utile réalisant la translation bidirectionnelle entre le codage TLV et la structure de donnée. Cette démarche se justifie par le fait que ces fonctionnalités sont plus importantes, dans un premier temps, dans le cadre du développement du système de messagerie Vidéomail Service, dans lequel le module de transcodage est susceptible d'intervenir. Les fonctionnalités de saisie et de visualisation pourront être réalisées par la suite.

Tous les modules, sauf le module d'entée/sortie et ceux relatifs à la saisie et la visualisation, ont été implémentés complètement. Une primitive d'analyse et de production des types définis dans la spécification a été réalisée pour chacun des types proposés dans l'exemple du point 5.2. Nous nous sommes limités à quelques primitives significatives, étant donné que nous disposerons d'un outil de génération, permettant de les produire automatiquement dans le cadre de n'importe quelle spécification.

Des tests ont été réalisés pour chacun des modules implémentés. L'architecture étant modulaire, ils ont pu être effectués indépendamment pour chacun des modules.

L'implémentation a été réalisée en langage C. Le choix s'est porté sur ce langage, parce qu'il permet le développement aisé d'applications portables. D'autre part, il permet à la fois des manipulations "proches de la machine" et la programmation de haut niveau.

5.5.1 Implémentation de la structure de donnée

Nous pouvons maintenant préciser les règles d'établissement de la structure de donnée, évoquées au point 5.4.1, étant donné que nous pouvons les exprimer en fonction des possibilités de structuration offertes par le langage C.

L'assignation d'un identificateur à une structure de donnée est effectuée par la déclaration typedef. Ainsi, pour signaler que l'identificateur std_identificateur fait référence à la définition de structure de donnée std_définition, on utilise la déclaration suivante :

```
typedef std_définition std_identificateur;
```

Par ailleurs, nous imposons à une structure de donnée d'être de longueur fixe. Ceci permet de diminuer la complexité de la structure et de simplifier les traitements qui y ont trait.

Lorsque le type de base d'une définition est constructeur, certains problèmes peuvent se poser. On peut spécifier le type d'un élément ou d'un membre en donnant l'identificateur d'un autre type défini dans la spécification ou d'un type pré-défini primitif. On peut également mentionner un type pré-défini constructeur. Cette dernière possibilité a été utilisée dans la spécification du dernier membre du type PersonnelRecord défini au point 5.2.

Comme nous le verrons plus loin, les primitives d'analyse et de production d'unités de donnée de protocole peuvent être fortement simplifiées lorsque les types des éléments ou des membres sont définis en citant seulement un identificateur, référençant soit un type pré-défini primitif, soit un autre type défini dans la spécification.

Nous devons donc modifier la spécification de PersonnelRecord de la manière suivante :

```
PersonnelRecord  ::= [APPLICATION 0] IMPLICIT SET {
                         Name,
                         title [0] IA5String,
                         EmployeeNumber,
                         dateOfHire [1] Date,
                         nameOfSpouse [2] Name OPTIONAL,
                         [3] IMPLICIT Auxiliaire
                               DEFAULT {}  }

Auxiliaire       ::= SEQUENCE OF ChildInformation
```

Cette spécification et celle donnée au point 5.2 sont équivalentes, si ce n'est qu'ici le type SEQUENCE OF ChildInformation a une identité propre, et porte le nom Auxiliaire.

Nous présenterons successivement les structures de donnée associées aux types pré-définis primitifs, aux types dont la définition utilise un type de base élémentaire, et

enfin aux types définis à partir d'un type de base constructeur.

5.5.1.1 Structure de donnée pour les types pré-définis primitifs

Nous avons vu au point 5.4.1.1 qu'aux types pré-définis BOOLEAN, INTEGER, OCTET STRING et BIT STRING correspondent des rerésentations propres. Celles-ci peuvent être établies une fois pour toutes, étant donné que ces types sont les briques de base du langage X409.

BOOLEAN

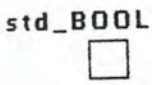
La structure de donnée que nous proposons pour le type BOOLEAN est l'octet. Celui-ci suffit pour représenter les valeurs de vérité, et est la plus petite unité de mémoire aisément accessible.

La valeur FALSE est représentée, tout comme dans le codage TLV, par un octet nul, la valeur TRUE étant équivalente à toute autre configuration binaire.

La structure de donnée choisie porte le nom std_BOOLEAN et est déclarée de la manière suivante :

```
typedef char std_BOOLEAN;
```

La représentation graphique de cette structure est la suivante :



INTEGER

Un problème se pose lors du choix de la structure de donnée pour le type INTEGER. On ne peut en effet pas toujours déterminer la valeur maximale que peut prendre l'entier.

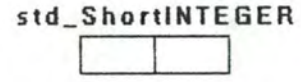
C'est pourquoi nous proposons deux représentations distinctes, la première étant applicable lorsque l'on

connait à l'avance la valeur maximale de l'entier, la seconde dans les autres cas.

Pour un type INTEGER {low(0),medium(1),high(2)}, la valeur maximale est connue à l'avance et vaut 2. On peut donc proposer la représentation de l'entier disponible dans le langage C. Nous l'appellerons dans ce qui suit std_ShortINTEGER, et sa déclaration prend la forme suivante :

```
typedef int std_ShortINTEGER;
```

Si l'entier disponible dans un environnement matériel déterminé occupe deux octets, la représentation graphique de cette structure est la suivante :

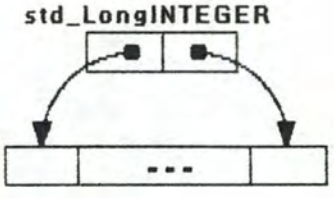


Pour un type INTEGER, la valeur maximale ne peut être connue à l'avance. Il est donc nécessire de proposer une autre représentation. Nous l'appellerons dans ce qui suit std_LongINTEGER, et sa déclaration est la suivante :

```
typedef struct
          {
             char * premier_octet;
             char * dernier_octet;
          } std_LongINTEGER;
```

L'entier est représenté par un ensemble d'octets consécutifs, délimité par premier_octet et dernier_octet.

La représentation graphique de cette structure de donnée est la suivante :

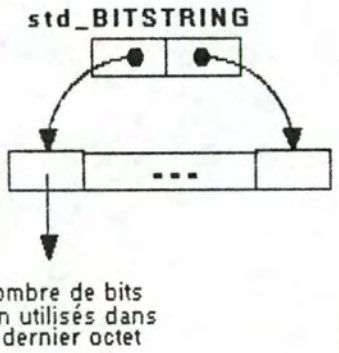


BIT STRING

La représentation que nous proposons pour les valeurs du type BIT STRING est un ensemble d'octets consécutifs, dont le premier indique le nombre de bits non utilisés dans le dernier octet de l'ensemble. Nous appellerons dans ce qui suit cette structure std_BITSTRING, et sa déclaration est la suivante :

```
typedef struct
        {
           char * premier_octet;
           char * dernier_octet;
        } std_BITSTRING;
```

La représentation graphique de cette structure de donnée est la suivante :

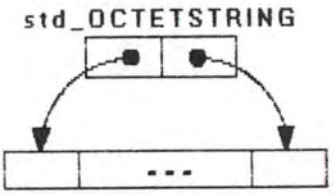


OCTET STRING

La représentation que nous proposons pour les valeurs du type OCTET STRING est un ensemble d'octets consécutifs. Nous appellerons dans ce qui suit cette structure std_OCTETSTRING, et sa déclaration est la suivante :

```
typedef struct
        {
           char * premier_octet;
           char * dernier_octet;
        } std_OCTETSTRING;
```

La représentation graphique de cette structure de donnée est la suivante :



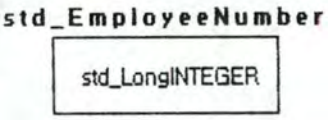
5.5.1.2. Structure de donnée pour les types définis à partir d'un type de base élémentaire

Comme nous l'avons vu au point 5.4.1.2, la structure de donnée choisie pour le type défini est strictement la même que celle représentant les valeurs du type de base.

Pour définir la structure de donnée associée au type EmployeeNumber, dont le type de base est élémentaire, nous utiliserons la déclaration suivante :

```
typedef std_LongINTEGER std_EmployeeNumber;
```

La représentation graphique de cette structure de donnée est la suivante :



5.5.1.3. Structure de donnée pour les types définis à partir d'un type de base constructeur

Nous avons vu au point 5.4.1.2 que les types prédéfinis constructeurs du langage X409 n'ont pas de représentation propre. En effet, la représentation effectivement choisie dépend de la spécification des éléments ou membres, dans le cas de la SEQUENCE ou du SET, et des types possibles de la valeur dans le cas du CHOICE.

Lorsque le type de base est la SEQUENCE ou le SET , on désire représenter un ensemble, ordonné ou non, de valeurs. Il y a lieu de distinguer les ensembles dont les éléments sont tous de même type (SEQUENCE OF et SET OF), mais présents en nombre variable, et les ensembles

contenant un nombre fixe d'éléments dont les types peuvent être différents (SEQUENCE { ... } et SET { ... }). Il est en effet nécessaire de trouver une structure de donnée de longueur fixe dans les deux cas.

- SEQUENCE OF / SET OF

Pour permettre à la structure de donnée associée aux types de base constructeurs SEQUENCE OF et SET OF d'être de longueur fixe, il est nécessaire de la définir comme liste chaînée. Un élément de cette liste est de longueur fixe, et l'ensemble complet est représenté par la référence du premier élément de la liste. Celle-ci est elle-même de longueur fixe.

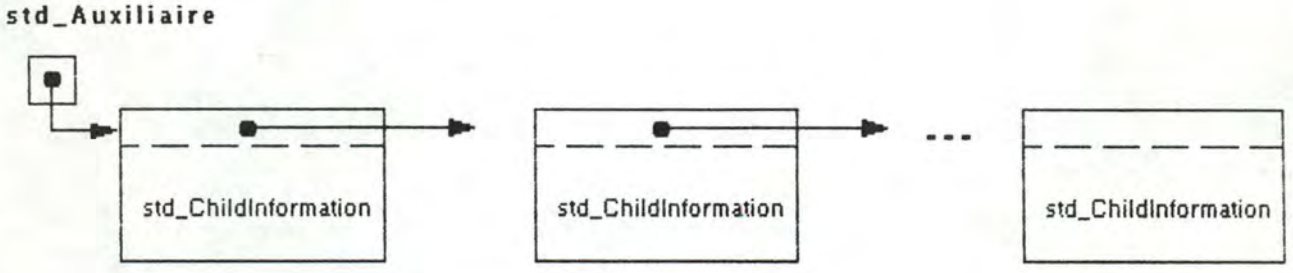
Ainsi, pour un élément de l'ensemble défini par le type Auxiliaire dégagé de la spécification du point 5.2, nous proposons la représentation suivante :

```
typedef struct S
        {
            struct S * réf_membre_suivant;
            std_ChildInformation  valeur_membre;
        } std_Membre_Auxiliaire;
```

La structure de donnée associée au type Auxiliaire lui même s'écrit alors :

```
typedef std_Membre_Auxiliaire * std_Auxiliaire;
```

La représentation graphique de la structure de donnée est la suivante :



L'ensemble est donc une liste chaînée d'éléments du type std_Membre_Auxiliaire. Il est représenté par un pointeur vers le premier élément de cette liste. Les représentations de l'ensemble et de ses membres sont donc de longueur fixe.

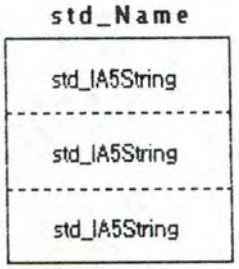
- SEQUENCE { ... } / SET { ... }

Les types de base constructeurs SEQUENCE { ... } et SET { ... } définissent un ensemble, ordonné ou non, d'éléments dont les types peuvent être différents, mais dont le nombre est connu à l'avance.

On peut donc proposer la représentation suivante pour le type Name défini dans la spécification du point 5.2 :

```
typedef struct S
        {
            std_IA5String  givenName;
            std_IA5String  initials;
            std_IA5String  familyName;
        } std_Name;
```

La représentation graphique de cette structure de donnée est la suivante :



Un problème se pose cependant lorsque certains des éléments de l'ensemble sont optionnels ou peuvent prendre des valeurs par défaut.

En effet, nous devons disposer d'un moyen permettant de signaler l'absence de la valeur de l'élément. Si nous considérons qu'elle n'est simplement pas représentée, nous ne pouvons pas garantir la taille fixe de la représentation.

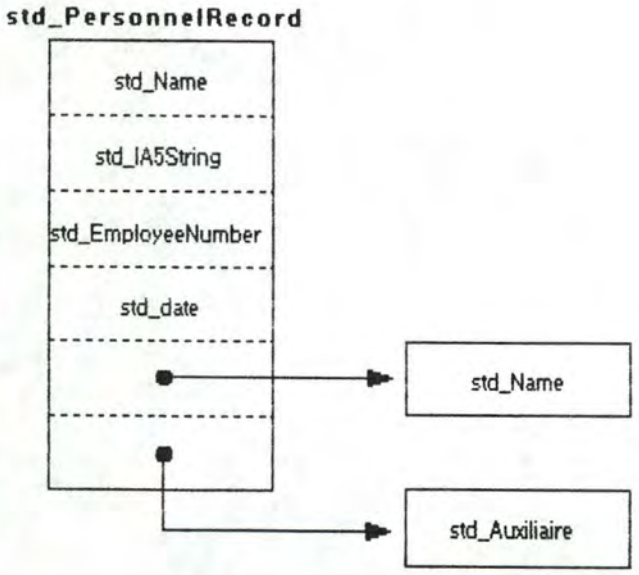
Une autre solution serait de réserver de la place dans la structure de donnée pour pouvoir y mettre la valeur de l'élément lorsque celle-ci existe, et qui contiendrait une valeur conventionnelle lorsque la valeur de l'élément est absente. Nous ne pouvons cependant pas déterminer cette valeur conventionnelle, étant donné que l'élément peut contenir a priori n'importe quelle information.

C'est pourquoi nous proposons un mécanisme d'indirection. L'ensemble contient pour chacun des éléments optionnels une référence à la représentation de sa valeur. Ainsi, lorsque la valeur existe, on peut l'obtenir par l'intermédiaire de cette référence. Dans le cas contraire, l'absence de la valeur est signalée par une référence vide.

La représentation de la structure de donnée proposée pour le type PersonnelRecord défini au point 5.2 est donc la suivante :

```
typedef struct
        {
            std_Name premier_membre;
            std_IA5String title;
            std_EmployeeNumber troisième_membre;
            std_Date dateOfHire;
            std_Name * nameOfSpouse;
            std_Auxiliaire * sixième_membre;
        } std_PersonnelRecord;
```

La représentation graphique est la suivante :



La valeur d'un élément optionnel (nameOfSpouse) est donc représenté par un pointeur vers la structure de donnée associée à Name. Lorsque la valeur de l'élément existe, le pointeur donne la référence de sa représentation. Dans le cas contraire, la valeur conventionnelle NULL indique que la valeur est absente. Ainsi, la structure de donnée associée à la représentation de l'ensemble garde une longueur fixe, quelle que soit la valeur des éléments optionnels.

- CHOICE

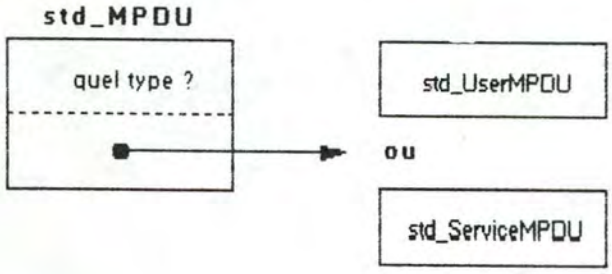
Dans le cas d'un type de base CHOICE, la structure de donnée doit pouvoir représenter une valeur d'un des types mentionnés dans la définition.

La structure de donnée contient deux parties. La première indique le type effectif de la valeur, et la seconde fournit une référence de la structure de donnée contenant la représentation de la valeur. Ainsi, la représentation est toujours de longueur fixe, quel que soit le type de la valeur.

Pour la définition d'une unité de donnée du protocole P1 (MPDU) dont la spécification a été présentée au point 3.3, nous proposons la représentation suivante :

```
typedef struct
        {
            int type_de_la_valeur;
             union
                 {
                  std_UserMPDU     *vers_valeur_UserMPDU;
                  std_ServiceMPDU
                                   *vers_valeur_ServiceMPDU;
                 } valeur;
        } std_MPDU;
```

La représentation graphique de cette structure de donnée est la suivante :



Le champ type_de_la_valeur donne simplement un numéro qui correspond à la position relative dans la spécification, du type. Ainsi, le type UserMPDU porte le numéro un, tandis que le type ServiceMPDU est identifié par le numéro deux.

5.5.2 Implémentation de l'architecture logicielle

Nous présenterons dans ce qui suit la mise en oeuvre de l'architecture logicielle proposée au point 5.4.2. Nous ne détaillerons que le module d'analyse des types définis dans la spécification et des types standards. Le fonctionnement des primitives de production est similaire. Les modules de saisie et de visualisation n'ont pas été implémentés.

L'analyse d'un élément de donnée TLV d'un certain type est toujours effectuée en trois étapes. Il faut d'abord étudier le composant T, et déterminer si les informations qu'il contient correspondent à la définition du type, en tenant compte des tags éventuels. Ensuite, on extrait la longueur codée dans le composant L, qui permet d'isoler le composant Valeur. Lorsque celui-ci est composé de plusieurs sous-éléments de donnée, il est nécessaire d'analyser chacun de ceux-ci. S'il est élémentaire, on peut interpréter l'information qu'il contient conformément à la définition du type de base.

Etant donné que le principe appliqué lors de l'analyse est toujours le même, on peut envisager d'utiliser des "macro-primitives" paramétrables pour réaliser ce traitement.

L'analyse du composant Valeur est cependant particulière, étant donné qu'il peut être composé de plusieurs sous-éléments, ou être élémentaire.

Dans le premier cas, l'analyse des différents sous-éléments doit être effectuée conformément aux règles appliquables au type de base constructeur utilisé. En effet, l'analyse ne s'effectue pas de la même manière si le type de base est la SEQUENCE, ou lorsqu'il est un type CHOICE.

Après avoir exposé brièvement les fonctionnalités des macro-primitives, nous présenterons quelques exemples d'utilisation.

Les macro-primitives que nous allons décrire sont des fonctions paramétrables effectuant le traitement relatif à l'analyse des composants T, L et V représentant le codage d'une valeur d'un type défini dans la spécification.

Etant donné que l'étude du composant V dépend du type de base utilisé dans le définition, nous distinguerons quatre types de macro-primitives. La première permet l'analyse lorsque le type de base est élémentaire. La seconde est utilisée dans le cas d'un type de base SEQUENCE { ... } ou SET { ... }, tandis que la troisième est d'application

lorsque le type de base est SEQUENCE OF ou SET OF. La dernière macro-primitive est utilisée dans le cas d'un type de base CHOICE.

- Macro-primitives pour un type de base élémentaire

La macro-primitive applicable dans le cas d'un type de base élémentaire permet d'isoler dans l'ensemble d'octets à analyser, le sous-ensemble correspondant au codage TLV d'une valeur du type de base. Nous l'appellerons dans ce qui suit etmac (Elementary base Type MACro-primitive).

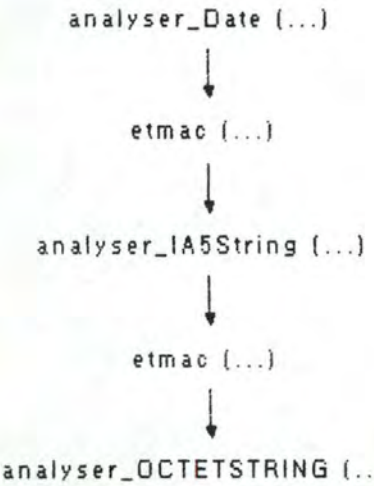
Pour ce faire, elle analyse les octets précédant ce sous-ensemble et représentant le codage TLV des tags éventuels.

Lorsque le sous-ensemble est isolé, elle invoque la primitive d'analyse associée au type de base. Celle-ci initialise la structure de donnée résultat avec l'information codée dans ce sous-ensemble.

Etant donné que les structures de donnée associées au type défini et au type de base sont identiques, la macro-primitive peut retourner la structure de donnée à la primitive qui l'a invoquée sans la modifier.

Les primitives d'analyse invoquant cette macro-primitive doivent lui fournir des informations au sujet des tags éventuels dont elle doit tenir compte, et mentionner quelle primitive est associée au type de base.

L'analyse d'un ensemble d'octets représentant la valeur du type Date défini au point 5.2 implique donc les appels suivants :



- Macro-primitives pour un type de base SEQUENCE { ... } ou SET { ... }

La macro-primitive applicable dans le cas d'un type de base SEQUENCE { ... } ou SET { ... } permet d'isoler dans l'ensemble d'octets à analyser, le sous-ensemble correspondant au codage du composant V du type de base. Nous l'appellerons dans ce qui suit ssmac (Sequence/Set base type MACro-primitive).

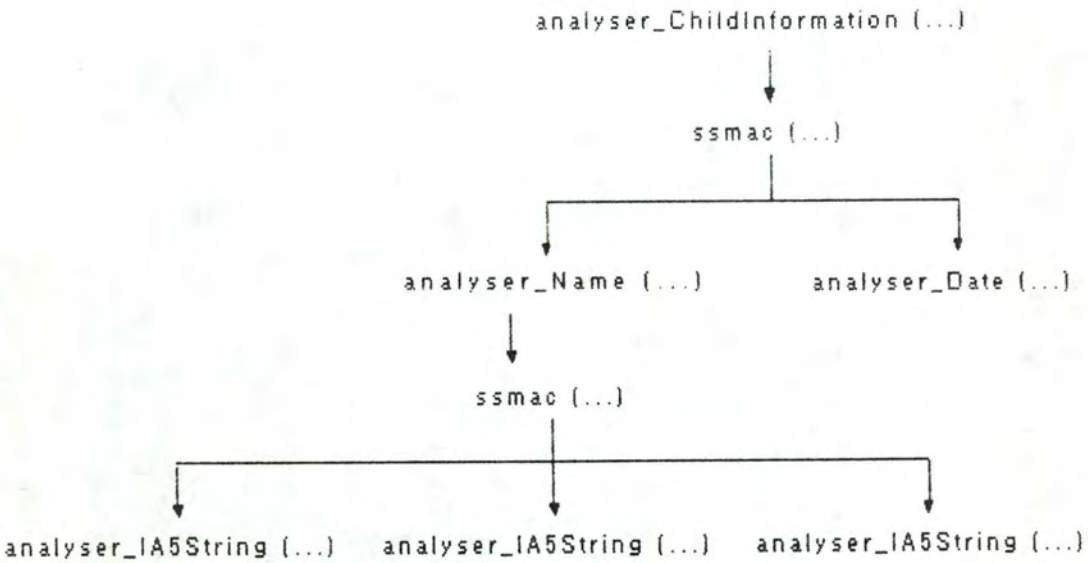
Pour ce faire, elle analyse les octets situés avant ce sous-ensemble, et représentant le codage des composants T et L du type de base, précédés du codage des tags éventuels.

Lorsque le sous-ensemble est isolé, elle invoque toutes les primitives associées à l'analayse du codage TLV des valeurs des types des éléments ou des membres. La succession des appels est déterminée par le type de base, étant donné que dans le cas de la SEQUENCE l'ordre d'apparition du codage des éléments est significatif, tandis qu'il ne l'est pas pour le SET.

Chaque fois d'une des primitives invoquées a analysé le codage d'un élément ou d'un membre, la macro-primitive garni la partie de la structure de donnée résultat qui y correspond avec la valeur retournée par celle-ci.

Les primitives d'analyse invoquant la macro-primitive ssmac lui donnent des informations au sujet des tags éventuels dont elle doit tenir compte. Elles fournissent également, pour chaque élément ou membre, des indications concernant les tags et la primitive associée à l'analyse d'une valeur de celui-ci.

L'analyse d'un ensemble d'octets représentant une valeur du type ChildInformation défini au point 5.2 implique donc les appels suivants :



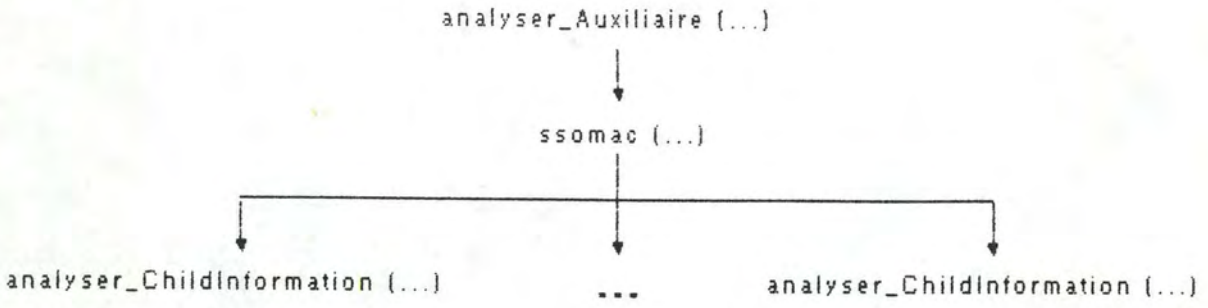
- Macro-primitives pour un type de base SEQUENCE OF ou SET OF

La macro-primitive applicable dans le cas d'un type de base SET OF ou SEQUENCE OF fonctionne de la même manière que la macro-primitive relative aux types SEQUENCE { ... } ou SET { ... }.

Elle se distingue cependant par l'analyse du sous-ensemble d'octets contenant le codage du composant V. Une seule primitive d'analyse est spécifiée, étant donné que les éléments sont tous de même type.

La primitive d'analyse est simplement invoquée autant de fois que possible, et la structure de donnée résultat est complétée au fur et à mesure.

L'analyse d'un ensemble d'octets représentant une valeur du type Auxiliaire dégagé de la spécification du point 5.2 implique donc les appels suivants :

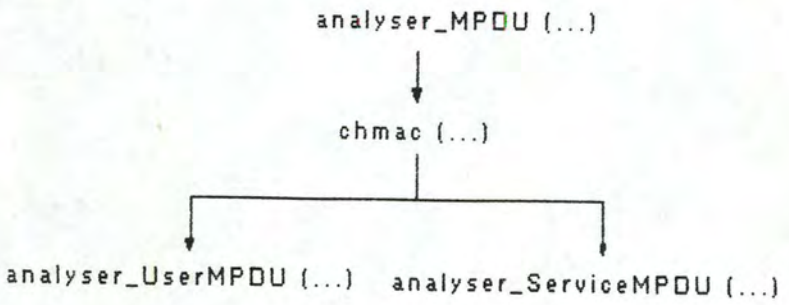


- Macro-primitives pour un type de base CHOICE

La macro-primitive applicable dans le cas d'un type de base CHOICE fonctionne de la même manière que celle au type de base élémentaire. Elle porte le nom chmac (CHoice base type MACro-primitive).

Une fois le sous-ensemble d'octets isolé, elle appelle successivement les primitives d'analyse associées aux différents types spécificiés dans la définition. Dès qu'une de ces primitives détermine que le sous-ensemble est le codage d'une valeur du type associé à cette primitive, la macro-primitive initialise la structure de donnée résultat avec la représentation obtenue.

L'analyse d'un ensemble d'octets représentant le codage TLV d'une valeur du type MPDU défini au point 3.3 implique les appels suivants :



L'annexe B contient le code en langage C relatif aux déclarations de structure de donnée et aux primitives d'analyse de quelques types définis dans la spécification du point 5.2.

6. Généralisation de l'outil d'analyse et de production d'unités de données de protocole

6.1. Introduction

L'outil d'analyse et de production d'unités de donnée de protocole, présenté aux chapitres 4 et 5 a été développé dans le cadre d'une spécification particulière. Nous avons mis en évidence la marche à suivre pour déduire de cette spécification les déclarations de structure de donnée et le code réalisant les fonctionnalités de transcodage.

Les principes proposés peuvent être appliqués à n'importe quelle spécification. Il est dès lors intéressant de concevoir un outil supplémentaire mettant en oeuvre ces principes dans le cadre d'une spécification quelconque. On pourrait ainsi automatiser l'élaboration de la structure de donnée et du code de transcodage.

De plus, cet outil pourrait effectuer certaines vérifications sur le texte formel, de manière à établir si la spécification qu'il représente est correcte.

Nous justifierons au point 6.2 les avantages de l'utilisation d'un générateur automatique par rapport à une élaboration manuelle de l'outil de transcodage.

Le point 6.3 est consacré à la spécification de ce générateur automatique, dont nous présenterons la mise en oeuvre au point 6.4.

Enfin, nous présenterons au point 6.5 quelques extensions envisageables pour cet outil, de manière à inclure par exemple le concept de "macro" (cf. point 3.3.2.3).

6.2. Justification de l'outil de génération automatique

L'outil d'analyse et de production d'unités de donnée de protocole a été conçu et réalisé de manière à minimiser les dépendances par rapport à une spécification particulière.

Nous avons vu qu'un grand nombre de modules sont réutilisables, étant donné que leurs fonctionnalités sont indépendantes de la spécification. Ainsi, les traitements relatifs aux composants T, L et V ne font intervenir que des concepts liés à la représentation concrète proposée par la recommandation X409. Ceux-ci sont invariants, quelle que soit la spécification donnée. De même, les fonctionnalités liées aux types pré-définis du langage sont applicables dans toute spécification, étant donné que le langage ne change pas lorsque la spécification est modifiée.

Les modules tributaires de la spécification des unités de donnée de protocole sont également conçus de manière à permettre une mise à jour aisée. Ainsi, un ensemble de primitives réutilisables, que nous avons appelées macro-primitives, sont disponibles.

Bien entendu, les mises à jour peuvent être apportées manuellement. L'effort à fournir est minimal étant donné que la taille des primitives à modifier est modeste.

Cette façon de procéder n'exclu cependant pas le risque d'erreurs. Il se peut que certaines informations extraites de la spécification des unités de donnée de protocole ne soient pas traduites correctement dans les primitives d'analyse et de production, ou dans la structure de donnée.

Il est donc possible que des erreurs s'introduisent dans le logiciel de transcodage. Les erreurs "évidentes" pourront être détectées assez rapidement, en procédant à des tests. Ceux-ci ne sont cependant pas exhaustifs, et rien ne permet d'affirmer que le logiciel modifié fonctionne correctement. D'autres erreurs ne pourront donc être détectées qu'en cours d'utilisation.

De plus, si le langage X409 intervient dans la spécification d'un programme d'application (cf. point 4.4.1), il est probable que le texte de la spécification soit modifié souvent. Les primitives de transcodage seraient donc à revoir périodiquement.

Lorsque la spécification est la description d'unités de donnée de protocoles normalisés, les changements sont plus rares. Ainsi, le logiciel de transcodage ne ferait l'objet de modifications que lorsqu'une nouvelle version de la spécification est publiée.

D'autre part, on n'est nullement assuré de la validité du texte formel utilisé. En effet, certaines incohérences risquent d'échapper à l'attention de la personne qui rédige la

spécification ou du programmeur réalisant les modifications du logiciel de transcodage.

Lorsque la spécification utilisée est tirée de recommandations officielles, comme c'est le cas pour les protocoles P1 ou P2 de la messagerie électronique X400, on peut supposer que le texte formel ne contient pas d'erreurs.

Ceci ne peut cependant pas être assuré pour une spécification intervenant dans le cadre de la conception d'une application. En effet, elle est en continuelle évolution et fait l'objet de modifications fréquentes. On ne peut donc être certain de sa validité à chaque instant.

6.3. Spécification de l'outil de génération automatique

Les fonctionnalités de l'outil de génération sont représentées à la figure 6.1. Le générateur prend en entrée une suite de caractères qui constitue le texte de la spécification. Il produit en sortie la déclaration de la structure de donnée et le code associé aux primitives d'analyse, de production, de saisie et de visualisation. Ces sorties sont des textes "sources" de programmes en langage C.

Il est à noter que nous n'avons pas tenu compte, dans le cadre de la mise en oeuvre de cet outil, de la possibilité de définition de notations non-standard de types et de valeurs. Celle-ci est liée au concept de macro, évoqué brièvement au point 3.3.2.3.

Le générateur que nous proposons ici ne permet de traiter que les notations standards de types et de valeurs.

Lors de l'analyse du texte en entrée, on doit tout d'abord vérifier si la spécification respecte la grammaire du langage X409. Cette validation porte essentiellement sur des aspects syntaxiques, et permet de déterminer si les notations de types et de valeurs présentes dans la spécification sont rédigées conformément aux règles du langage.

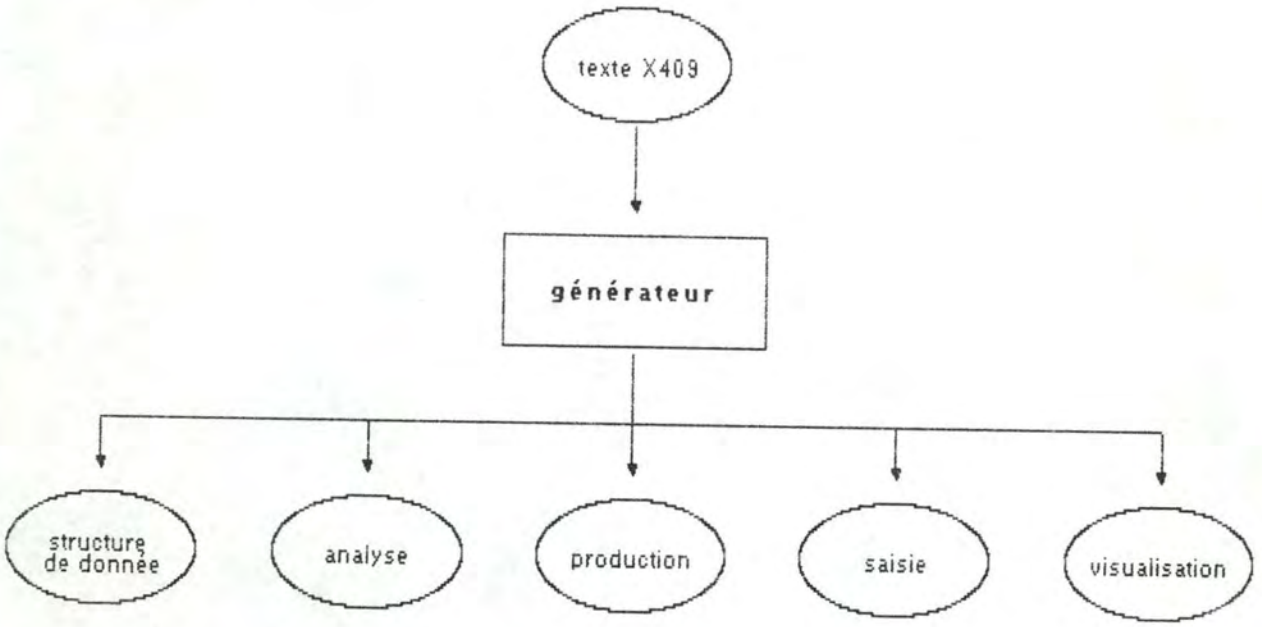


Figure 6.1

Fonctionnalités de l'outil
de génération automatique

Ainsi, la spécification suivante ne respecte pas la grammaire X409, étant donné qu'on ne peut la déduire de celle-ci par aucune combinaison d'applications successives de règles :

```
Erreur_Syntaxique DEFINITIONS ::=
  BEGIN
       Type1 := INTEGER
  END
```

Le respect strict de la grammaire ne suffit cependant pas pour affirmer que la spécification est correcte. En effet, l'exemple suivant est une instance correcte du langage :

```
Erreur_Sémantique DEFINITIONS ::=
  BEGIN
       Type1 ::= [0] IMPLICIT Type2
       Type1 ::= INTEGER
  END
```

Cependant, elle contient certaines incohérences. En effet, le type Type1 est défini deux fois. De plus, la première définition contient un tag de la classe CONTEXT, qui ne peut être mentionné qu'à l'intérieur d'un type constructeur. Enfin, aucune définition du type de base Type2 n'apparaît dans la spécification.

On constate que le texte d'une spécification, pour être cohérent, doit respecter des règles syntaxiques, énoncées par le biais d'une grammaire, et des contraintes d'ordre sémantique. Ces dernières ne sont pas exprimées dans la grammaire, et sont généralement données "en commentaires".

Lorsque le texte est jugé correct, le générateur peut, à l'aide des informations extraites de la spécification, produire la définition de la structure de donnée et générer le code réalisant les fonctionnalités de transcodage. Cette étape représente l'automatisation de la démarche proposée au chapitre 5 dans le cadre d'un exemple.

6.4. Réalisation de l'outil de génération automatique

Nous présenterons ici la mise en oeuvre de l'outil de génération dont les fonctionnalités ont été évoquées au point 6.3. Cette implémentation doit respecter un certain nombre de contraintes que nous évoquerons au point 6.4.1. L'architecture logicielle fait l'objet du point 6.4.2.

Etant donné que seul un sous-système utile de l'outil d'analyse et de production a été réalisé, nous n'avons pas intégré dans l'implémentation du générateur la notation de valeur disponible dans le langage X409. Cette démarche se justifie puisque cette notation n'intervient pas dans les fonctionnalités d'analyse et de production d'unités de donnée de protocole. Cette composante du langage peut être ajoutée par la suite.

6.4.1 Contraintes à respecter lors de la mise en oeuvre

La mise en oeuvre de l'outil de génération reprend essentiellement les deux étapes évoquées précédemment. Le texte de la spécification est analysé, et les informations qu'il contient sont ensuite exploitées pour produire la sortie.

L'analyse de la spécification doit être efficace. On peut envisager que la suite de caractères constituant le texte en entrée soit parcourue de gauche à droite, en prenant soin d'éviter les retours en arrière.

Si possible, les validations sont effectuées pendant ce parcours. Pour les vérifications syntaxiques, ceci ne pose pas de problèmes. En effet, lors de l'analyse de la suite de caractères :=, on peut déterminer qu'elle ne saurait faire partie d'une instance correcte du langage :

```
Exemple DEFINITIONS :=
  BEGIN
  END
```

Certaines validations sémantiques peuvent s'effectuer lors du parcours, d'autres ne sont réalisables que lorsque le texte complet a été analysé. Considérons l'exemple suivant :

```
Exemple DEFINITIONS ::=
  BEGIN
       Type1 ::= [0] IMPLICIT Type2
       Type1 ::= INTEGER
  END
```

On peut en effet affirmer lors de l'analyse de la suite de caractères [0] que le tag CONTEXT 0 qu'elle spécifie ne peut pas se trouver dans cette définition. De même, lors de l'étude de l'identificateur Type1 de la seconde définition, on peut affirmer que celui-ci a déjà fait l'objet d'une déclaration. Cependant, on ne peut déterminer que le type de base Type2 n'existe pas, que lorsque la spécification complète a été parcourue. Ceci est dû essentiellement à la possibilité qu'offre le langage X409 de définir les types et les valeurs dans un ordre quelconque. En effet, si le langage imposait de définir un identificateur avant son utilisation comme type de base, on pourrait affirmer lors de l'analyse de Type2 qu'aucune définition ne lui correspond.

Nous avons préféré scinder ces deux étapes, et reporter toutes les validations sémantiques après le parcours complet de la spécification. Ainsi, l'analyse syntaxique doit garnir une structure de donnée susceptible de représenter toutes les informations contenues dans la spécification. C'est sur cette structure de donnée que la validation sémantique est effectuée. De toute manière, elle sert de support d'information pour la phase de génération. Nous l'appellerons dans ce qui suit structure de donnée syntaxique.

Il faut cependant assurer que les messages d'erreur éventuels puissent guider valablement le spécificateur lors de la correction. Ainsi, on lui indique de manière précise où se trouve l'erreur, et de quelle incohérence il s'agit. Ceci implique que la structure de donnée contienne des informations sur la localisation dans le texte (par exemple un numéro de ligne), des informations qu'elle contient.

Une contrainte supplémentaire est imposée par le langage de programmation choisi pour le code généré. En effet, les identificateurs on généralement une taille limitée. Etant donné que le langage X409 permet l'utilisation d'identificateurs de longueur quelconque, il est nécessaire d'adapter ceux-ci aux conventions du langage de programmation. D'autre part, certains caractères (notamment le tiret "-") peuvent être présents dans un identificateur du langage X409, mais sont interdits dans les identificateurs du langage cible. Le générateur fait donc correspondre à chaque identificateur un nom qui respecte ces conventions. Nous l'appellerons dans ce qui suit un synonyme.

D'autre part, la production de la structure de donnée résultat soulève un problème. Il n'est pas toujours possible de déduire de la spécification, les identificateurs à utiliser dans les déclarations. Considérons l'exemple suivant :

```
Exemple DEFINITIONS ::=
  BEGIN
       Type1 := SEQUENCE
                {
                     phone INTEGER,
                           BOOLEAN
                }
  END
```

Nous avons vu qu'au type Type1 correspond la déclaration suivante :

```
typedef struct
        {
           std_LongINTEGER synonyme_phone;
           std_BOOLEAN nom_généré;
        } std_Type1;
```

Ainsi, le nom du premier champ peut être déduit directement de la spécification, en lui faisant subir certaines transformations. L'identificateur du second champ doit être généré. Il est nécessaire que le nom ainsi produit intègre suffisamment d'informations sur le contexte dans lequel il apparaît, sans quoi il risque d'être modifié lors d'une mise à jour, même minimale, de la spécification. Les conséquences sont graves, puisque la structure de donnée intervient directement dans la programmation de l'application. Considérons l'exemple suivant :

```
Exemple DEFINITIONS ::=
  BEGIN
       Type1 := SEQUENCE
                    {
                       INTEGER,
                       BOOLEAN
                    }
       Type2 := SEQUENCE
                    {
                       BOOLEAN,
                       OCTET STRING
                    }
  END
```

On peut proposer la structure de donnée suivante, qui utilise des noms de champs générés automatiquement, mais n'intégrant pas d'informations au sujet du contexte :

```
typedef struct
         {
            std_LongINTEGER généré_001;
            std_BOOLEAN généré_002;
         } std_Type1;

typedef struct
         {
            std_BOOLEAN généré_003;
            std_OCTETSTRING généré_004;
         } std_Type2;
```

On constate que si les définition associées à Type1 et Type2 sont inversées dans la spécification, les déclarations sont modifiées :

```
typedef struct
         {
            std_BOOLEAN généré_001;
            std_OCTETSTRING généré_002;
         } std_Type2;

typedef struct
         {
            std_LongINTEGER généré_003;
            std_BOOLEAN généré_004;
         } std_Type1;
```

Ceci signifie que le programmeur de l'application doit modifier toutes les primitives qui utilisent ces déclarations, alors que la spécification est elle-même "inchangée".

Lorsque les noms tiennent compte du contexte auquel ils ont trait, le risque de modifications est moins important. Ainsi, la déclaration suivante ne sera pas modifiée :

```
typedef struct
        {
           std_LongINTEGER type1_Premier_élément;
           std_BOOLEAN type1_Second_élément;
        } std_Type1;

typedef struct
        {
           std_BOOLEAN type2_Premier_élément;
           std_OCTETSTRING type2_Second_élément;
        } std_Type2;
```

Nous proposons, pour permettre l'indépendance complète du programmeur par rapport aux noms générés, un mécanisme d'indirection. Il a ainsi la possibilité de faire correspondre aux identificateurs (synonymes ou noms générés), des noms qu'il choisit lui-même. La correspondance peut s'établir de la manière suivante :

```
#define nom_choisi    nom_généré_ou_synonyme;
```

Ainsi, si des modifications d'intitulés interviennent dans la structure de donnée, seules ces déclarations ont à revoir.

6.4.2 Architecture logicielle

L'architecture logicielle que nous proposons pour la mise en oeuvre de l'outil de génération automatique est représentéé à la figure 6.2.

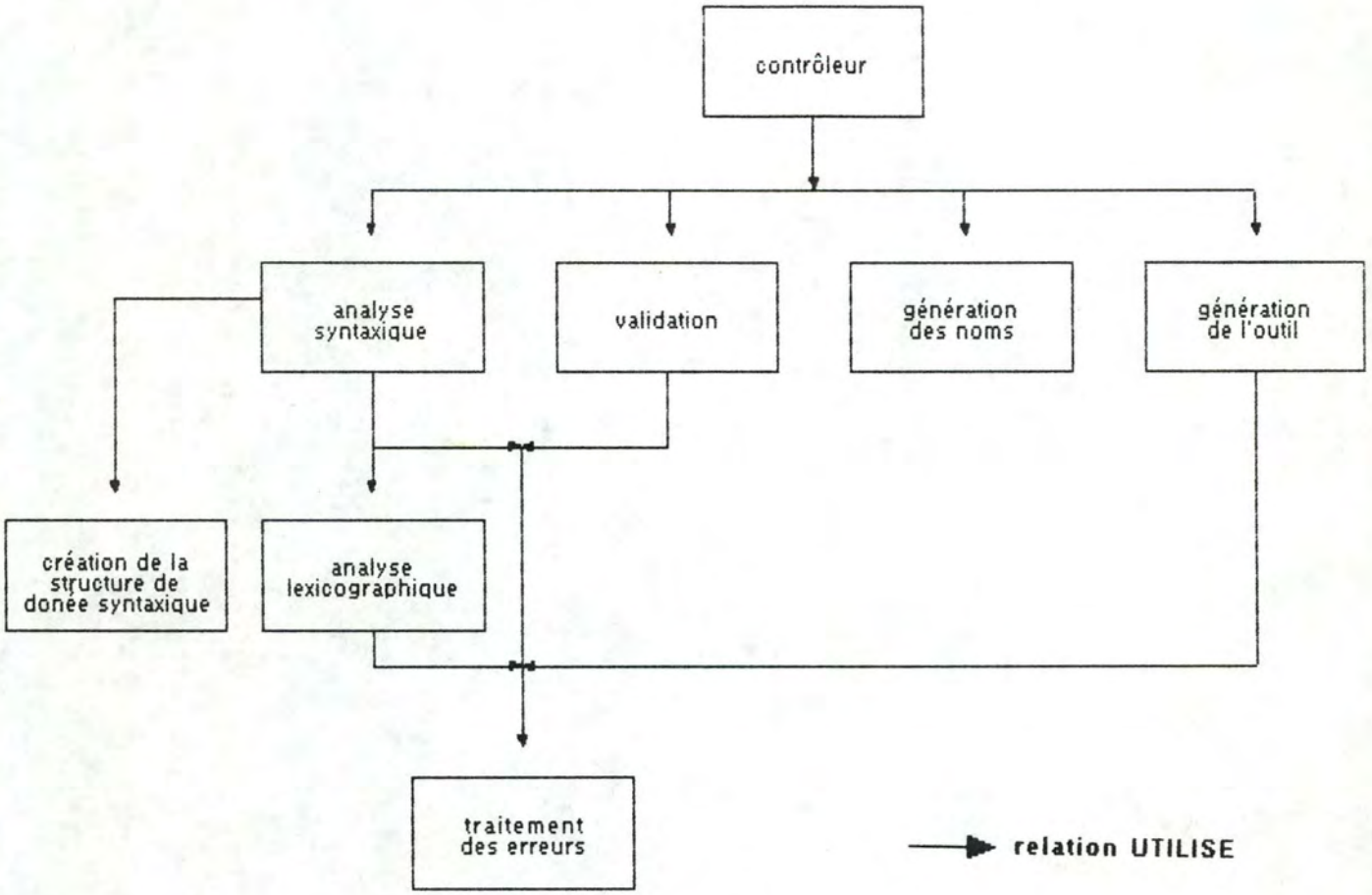


Figure 6.2

Architecture logicielle

6.4.2.1. Module contrôleur

Le module contrôleur permet d'enchaîner les étapes d'analyse et de vérification syntaxique, de validation sémantique et de génération.

6.4.2.2. Module d'analyse lexicographique

Le module d'analyse lexicographique permet d'isoler dans le texte à analyser des portions significatives, représentant par exemple des mots-clés du langage, des identificateurs, ...

Ce module a été réalisé à l'aide de l'utilitaire LEX de UNIX [30]. Celui-ci permet de générer une primitive d'analyse en langage C à partir d'un ensemble de règles lexicographiques.

Celles-ci sont des expressions régulières, et à chacune d'entre elles correspond un ensemble de chaînes de caractères. Considérons les deux expressions suivantes :

```
integer
[A-Z]+
```

La première expression régulière détermine l'ensemble constitué de la chaîne de caractères integer. La seconde définit un ensemble de suites d'un ou plusieurs (opérateur +) caractères correspondant à des lettres majuscules. Celles-ci appartiennent à la classe (opérateurs [et]) des caractères compris (opérateur -) entre A et Z.

Pour chacune de ces règles, on peut spécifier une action, que l'analyseur généré par LEX exécute lorsqu'une suite de caractères appartenant à l'ensemble défini par celle-ci a été détectée dans le flux en entrée. L'action est un bloc d'instructions rédigées en langage C. Nous pouvons ainsi compléter l'exemple précédent de la manière suivante :

```
integer    {
             printf ("mot-clé integer");
           }
[A-Z]+     {
             printf ("identificateur %s",yytext);
           }
```

La première action consiste à imprimer le commentaire "mot-clé integer" lorsque la chaîne de caractères integer a été détectée dans la suite de caractères en entrée.

La seconde est exécutée lorsqu'un identificateur constitué d'au moins une lettre majuscule se trouve en entrée. On peut faire référence à la chaîne reconnue par le biais de la variable yytext.

La primitive d'analyse générée par LEX peut être utilisée par d'autres modules pour réaliser l'entrée "de bas niveau". Elle signale aux primitives qui l'invoquent la présence d'une chaîne de caractères significative en leurs retournant une valeur numérique appelée jeton (token). Elle peut également transmettre une valeur quelconque par le biais d'une variable spéciale dénomée yyval. Celle-ci servira par exemple pour communiquer aux primitives invoquant l'analyseur, la chaîne de caractères détectée.

Nous renvoyons le lecteur désirant de plus amples informations à la référence [30]. L'annexe C contient la spécification LEX utilisée pour générer le module d'analyse lexicographique.

6.4.2.3. Module d'analyse syntaxique

Le module d'analyse syntaxique permet de déterminer si le texte à analyser représente une instance correcte d'un langage défini par le biais d'une grammaire. Ici, la grammaire est celle du langage X409.

Ce module a été entièrement réalisé grâce à l'utilitaire YACC (Yet Another Compiler Compiler) de UNIX [31]. Il permet de générer une primitive d'analyse en langage C à partir d'une grammaire définie dans le formalisme BNF.

Les règles qu'elle contient sont rédigées de la manière suivante :

$$\text{Symbole\_Non\_Terminal} : \text{Symbole}_1 \; \text{Symbole}_2 \; \ldots \; \text{Symbole}_n \;;$$

Cette règle indique que Symbole_Non_Terminal est équivalent à la suite des symboles terminaux ou non-terminaux Symbole_i (i : 1 .. n).

Les symboles terminaux correspondent aux jetons que peut renvoyer la primitive d'analyse lexicographique générée par LEX. Ils sont déclarés dans la spécification YACC de la manière suivante :

```
%token  nom_de_jeton
```

Les symboles non-terminaux font référence à d'autres règles, où ils sont définis de manière plus précise.

La primitive d'analyse générée par YACC est capable d'établir si le texte en entrée est une instance correcte d'un symbole non-terminal particulier, appelé symbole de démarrage. On le désigne de la manière suivante :

```
%start symbole_non_terminal
```

En présence d'un texte en entrée, la primitive tente de déterminer la combinaison d'applications successives de règles, qui permet d'affirmer si le texte est une instance correcte du symbole de démarrage. Chaque fois qu'une règle est applicable, la primitive peut exécuter une action, constituée d'un bloc d'instructions en langage C. On la définit de la manière suivante :

```
Symbole_Non_Terminal :    Symbole1
                          Symbole2
                            ...
                          Symbolen
                            {
                              action
                            }
                     ;
```

De plus, lorsqu'un symbole non-terminal est établi en vertu d'une des règles qui lui correspondent, ce symbole peut se voir attribué une valeur. Celle-ci est initialisée dans l'action associée à la règle, par le biais de la pseudo-variable "$$". D'autres actions peuvent ensuite l'exploiter. Il en est de même pour les symboles terminaux. La valeur est initialisée dans ce cas par la primitive d'analyse lexicographique, lorsqu'elle signale la présence du symbole terminal dans le texte en entrée. Considérons l'exemple suivant :

```
Non_Terminal_1        :    Terminal_A
                           Non_Terminal_2
                             {
                              $$ = fonction ($1,$2);
                             }
                      ;
```

```
Non_Terminal_2        :    Terminal_B
                             {
                              $$ = fonction ($1);
                             }
                      ;
```

L'action de la première règle spécifie que la valeur associée à Non_Terminal_1 ($$) est calculée "en fonction" des valeurs associées à Terminal_A ($1) et Non_terminal_2 ($2). Cette dernière valeur a été produite précédemment lors de l'application de la règle associée à Non_Terminal_2, en exécutant la seconde action.

Ainsi, il est possible d'exploiter les informations contenues dans le texte de spécification en entrée par le biais des actions. Elles contiennent des appels aux primitives du module de création de la structure de donnée, qui permettent de mémoriser cette information en vue d'un traitement ultérieur.

Les valeurs ainsi échangées entre les actions peuvent être de types quelconques. Les différents types possibles pouvant exister au sein de la spécification YACC sont définis comme suit :

```
%union
     {
       type1 champ1;
       type2 champ2;
        ...
       typen champn;
     }
```

Nous renvoyons le lecteur désirant de plus amples informations, notamment sur le principe de fontionnement de l'analyseur généré, à la référence [31]. L'annexe C contient la spécification YACC utilisée pour générer le module d'analyse syntaxique.

6.4.2.4. Module de création de la structure de donnée syntaxique

Le module de création de la structure de donnée syntaxique contient un ensemble de primitives destinées à créer et garnir la structure de donnée syntaxique.

Celle-ci contient, comme nous l'avons vu, les informations extraites de la spécification analysée, et sert de base pour la validation sémantique et la génération de code. Elle est garnie tout au long du

parcours de la spécification, par l'invocation des primitives du module de création.

6.4.2.5. Module de validation

Le module de validation exploite la structure de donnée syntaxique contenant les informations extraites de la spécification.

Son objectif est de vérifier que la spécification respecte les contraintes évoquées au point 3.3.2. Celles-ci ne sont pas exprimées dans la grammaire, mais ont été données en "commentaire".

6.4.2.6. Module de génération des noms

Le module de génération des noms permet d'attribuer des identificateurs de champs dans les déclarations de structure de donnée, lorsque ceux-ci ne peuvent être déduits directement de la spécification.

D'autre part, il détermine les synonymes pour chacun des identificateurs (générés ou déduits de la spécification) intervenant dans les déclarations de structure de donnée.

Ces synonymes respectent les conventions du langage de programmation utilisé, relatives aux identificateurs. Comme nous l'avons vu, ces conventions portent essentiellement sur la taille des identificateurs, et les caractères spéciaux qu'ils peuvent contenir.

Le module tente de conserver dans le synonyme tous les éléments relatifs au contexte dans lequel apparaît l'identificateur à modifier.

6.4.2.7. Module de génération de l'outil de transcodage

Le module de génération de l'outil de transcodage exploite la structure de donnée syntaxique pour produire l'outil proposé au chapitre 5.

Nous avons vu qu'il est nécessaire de déduire la structure de donnée dont disposeront les programmes d'application, et les primitives d'analyse et de production.

Nous ne détaillerons pas ces deux étapes, étant donné qu'elles représentent la simple automatisation de la démarche proposée au chapitre 5.

Une particularité mérite cependant une certaine attention. En effet, le langage de programmation impose un ordre particulier pour les déclarations de structure de donnée. Un identificateur doit avoir été défini avant son utilisation dans une autre déclaration.

Ainsi, les déclarations relatives aux types ChildInformation, Name et Date définis dans l'exemple du point 5.2 doivent être mentionnées dans l'ordre suivant :

```
typedef std_IA5String std_Date;

typedef struct
        {
          std_IA5String givenName;
          std_IA5String initials;
          std_IA5String familyName;
        } std_Name;

typedef struct
        {
          std_Name nom_généré;
          std_Date dateOfBirth;
        } std_ChildInformation;
```

L'ordre imposé ici n'a rien de commun avec celui des définitions de types dans la spécification.

L'ordre des déclaration peut cependant être aisément déterminé. Il suffit de considérer la structure de donnée syntaxique comme graphe, où chaque définition de type représente un noeud, et où les arcs établissent le lien entre le type défini et les définitions de type dont il dépend. Lorsque le type de base est élémentaire, il existe un arc vers la définition de ce type, sauf s'il s'agit d'un type de base pré-défini primitif. Dans le cas contraire, il existe un arc vers la définition de chacun des types des éléments, sauf si ceux-ci sont des types pré-définis primitifs. On peut alors envisager un parcours "en profondeur d'abord", au cours duquel on produit les déclarations de structure de donnée. Ainsi, on est assuré que les déclarations sont établies dans le bon ordre.

Il faut assurer, dans l'approche choisie ici, que ce graphe ne contienne pas de cycles. En effet, le parcours risque alors de ne pas se terminer. Les définitions récursives n'étant pas explicitement autorisées dans le cadre du langage X409, cette situation ne peut normalement pas se présenter. Si cette possibilité de définition

récursive de type de données était intégrée dans le langage lors de versions ultérieures, il sera nécessaire d'envisager d'autres approches que celle adoptée ici.

6.4.2.8. Module de traitement des erreurs

Le module de traitement des erreurs fournit un ensemble de primitives permettant d'afficher des messages d'erreur. Celles-ci sont utilisées par les modules d'analyse lexicographique et syntaxique, de validation et de génération de l'outil.

6.5. Extensions possibles pour le générateur

Nous n'avons pas considéré lors de la réalisation du générateur automatique de l'outil de transcodage, la possibilité offerte par le langage X409 de définir, à l'aide de la construction MACRO, de nouvelles règles de notation de types et de valeurs.

La spécification des règles de grammaire du langage X409 qui nous a permis de générer une primitive d'analyse syntaxique, est statique. En effet, les règles de notation standards ne changent pas d'une spécification à l'autre.

La primitive d'analyse générée par YACC en fonction de ces règles de grammaire n'est donc pas à même de tenir compte des nouvelles règles de notation mentionnées dans la spécification des unités de donnée de protocole qu'elle analyse.

D'autre part, la primitive générée par LEX ne peut reconnaître que des mots-clés correspondant aux expressions régulières spécifiées lors de la génération. Si les définitions de macros introduisent de nouveaux mots-clés, la primitive d'analyse lexicographique n'est pas à même de les détecter dans le texte en entrée.

Si l'on considère que les déclarations de macros modifient le langage, il est nécessaire de disposer de primitives d'analyse lexicographique et syntaxique pour ce nouveau langage.

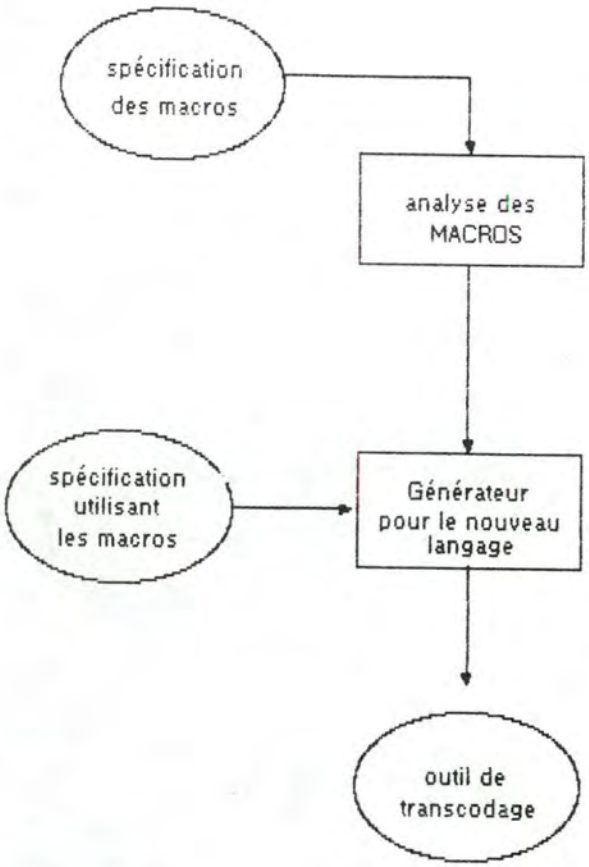


Figure 6.3

Inclusion des définitions de MACROs
dans la génération automatique

Nous proposons de considérer séparément les parties de texte spécifiant les définitions de macros et les définitions d'unités de donnée de protocole faisant intervenir des notations relatives à celles-ci. Ainsi, le texte contenant les déclarations de macros est analysé, pour déterminer comment les règles de grammaire du langage X409 doivent être modifiées, en vue de correspondre à ce nouveau langage. Ensuite, les modifications éventuelles sont intégrées dans les spécifications LEX et YACC. Il suffit alors de générer de nouvelles primitives d'analyse lexicographique et syntaxique pour analyser le texte utilisant les macros.

Ceci n'est qu'une ébauche de solution. Il faut en effet considérer si la structure de donnée syntaxique telle que nous l'avons utilisée est à même de représenter les informations relatives à des types de donnée définis à l'aide des macros.

La démarche que nous proposons pour intégrer les déclarations de macros dans le générateur est représentée à la

figure 6.3. On constate qu'il est nécessaire que le texte de spécification soit modifié manuellement, de manière à produire une spécification contenant uniquement des déclarations de macros, et un texte constitué seulement de définitions de types et de valeurs, faisant intervenir éventuellement les macros ainsi définies.

Conclusion

Un ensemble de standards internationaux permettent à des systèmes de games différentes ou de constructeurs distincts de collaborer pour assurer le transfert d'information. L'existence de tels standards est bénéfique, tant pour les utilisateurs que pour les fournisseurs de matériel et de logiciel informatique.

En effet, les propositions standardisées ne sont généralement pas rédigées de manière claire et précise, et peuvent bien souvent donner lieu à des interprétations fort diverses. Ceci est essentiellement dû à l'utilisation de la langue naturelle dans les recommandations officielles.

Pour réduire le risque d'interprétations erronnées, les organismes de standardisation proposent de plus en plus des spécifications rédigées dans des langages formels.

Leur formalisme, plus limité mais par le fait même moins ambigu que la langue naturelle, autorise d'autre part la conception et la mise en oeuvre d'outils automatisés manipulant des spécifications rédigées dans ce langage formel.

Cet aspect des langages formels de spécification peut s'avérer très avantageux pour les fournisseurs de logiciels distribués. Ils disposent d'une part de spécifications plus claires et moins ambiguës des propositions normalisées, et d'autre part d'outils automatisés susceptibles de réduire fortement le coût de développement d'une application distribuée relatives à ces standards.

Nous avons, dans le cadre de ce travail, développé un outil manipulant des spécifications rédigées dans le langage X409 proposé par le CCITT. Il peut générer un outil d'interface permettant d'exprimer une valeur d'une unité de donnée d'un protocole spécifié dans le langage X409, dans diverses représentations. Cet outil d'interface peut alors s'intégrer dans des logiciels d'application, qui disposent de ses services pour manipuler de l'information représentée dans un format "commode" et efficace.

Nous avons mis en évidence un certain nombre de domaines d'application, dans lesquels ces deux outils peuvent intervenir.

Nous nous sommes limités, lors de la réalisation de ceux-ci, aux règles de notation standards du langage X409, et nous n'avons pas abordé la notation de valeur qu'il propose.

Il serait intéressant d'inclure dans ces outils la possibilité de définition et d'utilisation de règles de notation non-standards de types et de valeurs. Ceci est important si l'on considère que cette composante du langage intervient déjà dans la spécification du protocole P3 de la messagerie électronique X400, et que ce concept de "macro" est susceptible de prendre une importance toujours croissante dans la spécification des protcoles développés à l'avenir.

D'autre part, il serait nécessaire de déterminer "sur le terrain" si la structure de donnée que nous avons proposée comme format de représentation pour les unités de donnée de protocole permet effectivement de simplifier la tâche du programmeur d'une application distribuée.

Bibliographie

[1] G. Pujolle et al., Réseaux et Télématique, tome I et tome II, Eyrolles, 1985

[2] Folts et al., Open Systems Interconnection, ITT Europe, 22-24 Mai 1985, Vol. I et II

[3] International Standards Organization (ISO), Information Processing Systems - Open Systems Interconnection - ESTELLE - A Formal Description Technique Based on an Extended State Transition Model, ISO/DP/9074, September 1986

[4] P. Bovy, S. Crasset, Protocoles de Communication : Etude des Outils de Vérification de Conformité d'une Implémentation, mémoire présenté en vue de l'obtention du titre de licencié et maître en informatique, Facultés Universitaires N.-D. de la Paix, Namur, 1986

[5] G. Leduc, Internet Service Verification, ESPRIT Project 73 Area 4.3, Université de Liège, Liège, Février 1985

[6] M. Diaz, SEDOS : Un Environnement Logiciel pour la Conception des Systèmes Distribués, 3x Congrès "De nouvelles architectures pour les communications", Paris 28-30 Octobre 1986, pp. 164-171

[7] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Terminal Equipement and Protocols for Telematic Services - Document Interchange Protocol for the Telematic Services - (Recommendation T73), Vol. VII, Fascicle VII.3, Geneva, 1985, pp. 453-508

[8] The OMNICOM Institute, Message Handling Systems : X400 Architecture and Protocols, Vienna, 1986

[9] International Standards Organization (ISO), Information Processing Systems - Open Systems Interconnection - Basic Reference Model, ISO/IS/7498

[10] R. Linn, The Features and Facilities of ESTELLE, National Bureau of Standards, Gaithersburg, Septembre 1986

[11] S. Boudkowski et al., ESTELLE - Un Langage de Spécification des Systèmes Distribués, 3º Congrès "De nouvelles architectures pour les communications", Paris 28-30 Octobre 1986, pp. 118-129

[12] International Standards Organization (ISO), Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO/DP/8807, July 1985

[13] G. Leduc, Assessing the Service Provided by a Connection-less Protocol, Protocol Specification, Testing and Verification V, North Holland, 1985

[14] E. Nam et al., LOTOS, Un nouveau Langage de Spécification des Systèmes Distribués, 3º Congrès "De nouvelles architectures pour les communications", Paris 28-30 Octobre 1986, pp. 130-161

[15] E. Brinksma, A Tutorial on LOTOS, Twente University of Technology, Enschede

[16] V. Carchiolo et al., A LOTOS Specification of the PROWAY Highway Service, IEEE Transactions on computers, Vol. C-35, nº 11, Novembre 1986

[17] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Information Processing Systems - OSI - Specification of Basic Encoding Rules for Abstract Syntax and Notation (ASN.1), Question 40/VII, Contribution tardive D311, Genève, Sept./Oct. 1986

[18] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Information Processing Systems - OSI - Specification of Abstract Syntax and Notation (ASN.1), Question 40/VII, Contribution tardive D310, Genève, Sept./Oct. 1986

[19] D. Redell et al., Interconnecting Electronic Mail Systems, Computer, Sept. 1983, pp. 53-63

[20] T. Myer, Standards For Global Messaging : A Progress Report, Journal of Telecommunication Networks, pp. 413-433

[21] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : System Model - Service Elements (Recommendation X400), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 3-38

[22] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Basic Service Elements and Optional User Facilities (Recommendation X401), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 39-45

[23] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Encoded Information Type Conversion Rules (Recommendation X408), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 45-61

[24] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Presentation Transfer Syntax and Notation (Recommendation X409), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 62-93

[25] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Remote Operations and Reliable Transfer Server (Recommendation X410), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 93-126

[26] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Message Transfer Layer (Recommendation X411), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 127-182

[27] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Interpersonnal Messaging User Agent Layer (Recommendation X420), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 182-219

[28] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Data Communication Networks - Message Handling Systems : Access Protocol for Teletex Terminals (Recommendation X430), Volume VIII - Fascicle VIII.7, Geneva, 1985, pp. 219-266

[29] B. Plattner et al., PROSPECT - A Tool for Protocol Specification and Conformance Testing, ETH Zürich, Zürich, 1987

[30] M. Lesk, LEX - A Lexical Analyzer Generator, Unix Programmers Manual, Seventh Edition, Volume 2A, January 1979

[31] S. Johnson, YACC - Yet Another Compiler Compiler, Unix Programmers Manual, Seventh Edition, Volume 2A, January 1979

[32] Comité Consultatif International pour la Télégraphie et la Téléphonie (CCITT), Near-term Extensions to ASN.1 (Version 1), Working Document, Question 40/VII, Geneva, October 1986

Liste d'abréviations

ADMD	:	ADministration Management Domain
ASN.1	:	Abstract Syntax and Notation One
CCITT	:	Comité Consultatif International pour la Téléphonie et la Télégraphie
EOC	:	End Of Contents
IAT	:	Implémentation A tester
IPMS	:	InterPersonal Messaging System
ISO	:	International Standards Organization
MD	:	Management Domain
MHS	:	Message Handling System
MPDU	:	Message Protocol Data Unit
MTA	:	Message Transfer Agent
MTAE	:	Message Transfer Entity
MTAL	:	Message Transfer Layer
MTS	:	Message Transfer System
OSI	:	Open Systems Interconnection
P1	:	Protocole entre MATEs paires
P2	:	Protocole entre UAEs paires de la messagerie IPMS
P3	:	Protocole entre MTAE et SDE paires
PDU	:	Protocol Data Unit
PRMD	:	PRivate Management Domain
P_c	:	Protocol for Contents
SAP	:	Service Access Point
SDE	:	Submission and Delivery Entity
SDU	:	Service Data Unit

SFD	:	Simple Formattable Document
ServiceMPDU	:	SERVICE Message Protocol Data Unit
TLV	:	Type Longueur Valeur
UA	:	User Agent
UAE	:	User Agent Entity
UAL	:	User Agent Layer
UMPDUContent	:	User Message Protocol Data Unit CONTENT
UMPDUEnvelope	:	User Message Protocol Data Unit ENVELOPE
UserMPDU	:	USER Message Protocol Data Unit
VMS	:	messagerie électronique VidéoMail Service

ANNEXES

Annexe A : Grammaire du langage X409

```
TypeDefinition    ::=  identifier "::=" Type

Type              ::=  PreDefinedType | DefinedType

PreDefinedType    ::=  BOOLEAN |
                       INTEGER |
                       INTEGER {NamedNumberList}
                       BIT STRING |
                       BIT STRING {NamedNumberList}|
                       OCTET STRING
                       NULL
                       SEQUENCE
                       SEQUENCE OF Type
                       SEQUENCE {ElementTypes}
                       SET
                       SET OF Type
                       SET {MemberTypes}
                       Tag IMPLICIT Type
                       Tag Type
                       CHOICE {AlternativeTypeList}

DefinedType       ::=  identifier

NamedNumberList   ::=  NamedNumber |
                       NamedNumberList , NamedNumber

NamedNumber       ::=  identifier (NumericValue)

NumericValue      ::=  number | DefinedValue

ElementTypes      ::=  OptionalTypeList | empty

MemberTypes       ::=  OptionalTypeList | empty

OptionalTypeList  ::=  OptionalType |
                       OptionalTypeList , OptionalType

OptionalType      ::=  NamedType | NamedType OPTIONAL
                       NamedType DEFAULT Value
                       ComponentsOf

NamedType         ::=  identifier Type | Type

ComponentsOf      ::=  COMPONENTS OF Type

AlternativeTypeList ::= NamedType | AlternativeTypeList ,
                          NamedType

Tag               ::=  [Class NumericValue]
```

```
Class              ::=  UNIVERSAL | APPLICATION |
                        PRIVATE | empty

ValueDefinition    ::=  identifier Type "::=" Value

Value              ::=  PreDefinedValue | DefinedValue

PreDefinedValue    ::=  TRUE | FALSE
                        number | - number |
                        identifier |
                        {IdentifierList}
                        bstring | cstring | hstring |
                        NULL |
                        {Values}
                        identifier Value
                        Type Value

DefinedValue       ::=  identifier

IdentifierList     ::=  identifier | IdentifierList ,
                        identifier

Values             ::=  NamedValueList | empty

NamedValueList     ::=  identifier Value | Value


ModuleDefinition   ::=  identifier DEFINITIONS "::="
                        BEGIN ModuleBody END

ModuleBody         ::=  DefinitionList | empty

DefinitionList     ::=  TypeDefinition | ValueDefinition
```

Annexe B : Exemples de primitives d'analyse

Les extraits de programmes qui suivent montrent que l'utilisation de macro-primitives d'analyse peuvent réduire sensiblement la taille du code à modifier lors du passage à une nouvelle spécification.

On y trouve tout d'abord les déclarations des structure de donnée, et ensuite le code associé aux primitives d'analyse pour deux types dont les définitions on été données au point 5.2. Il s'agit des types ChildInformation et Date. Les primitives relatives aux autres types sont similaires.

```
typedef std_IA5String std_Date;

typedef struct
        {
          std_Name childInformation_1;
          std_Date dateOfBirth;
        } std_ChildInformation;


/*-------------------------------------------------------------------------*/
/*        Primitive d'analyse pour le type ChildInformation                */
/*-------------------------------------------------------------------------*/

analyser_ChildInformation (opt,deb,fin,tagged,t_cc,t_i,new_deb,x)

  int opt;              /* optionel ?                 */
  ptv_Byte deb,fin;     /* délimiteurs de l'enseemble */
  int tagged;           /* tag supplémentaire ?       */
  int t_cc;             /* classe                     */
  long t_i;             /* identificateur             */

  ptv_Byte             *new_deb;     /* délimiteur du sous-ensemble résultat */
  std_ChildInformation *x;           / structure de donnée résultat         */


{

  int verdict,result;   /*   retour pour appels de primitives      */

    /*   primitives d'analyse pour les éléments */

  int analyser_Name ();
  int analyser_Date ();
```

```
  static EL_CTRL elements [] =   /* description des éléments   */
 {
  "Name" ,&analyser_Name,NOTAG   ,NULL   ,NULL,sizeof(std_Name),NULL,NO,NULL,
  "Date" ,&analyser_Date,EXPLICIT,CONTEXT,0L  ,sizeof(std_Date),NULL,NO,NULL,
  "\0"   ,NULL          ,NULL    ,NULL   ,NULL,NULL            ,NULL,NO,NULL

  );

/*   incrementation de la profondeur d'appel  de primitives*/
/*   ------------------------------------------------   */

  t_idl ("analyserChildInformation",deb,fin);

     /* adresses des éléments     */
    /* de la structure de donnée */

  elements[0].vadr = &(x->childInformation_1);
  elements[1].vadr = &(x->dateOfBirth);

  result = analyser_ssmac (opt,deb,fin,NOTAG,NULL,NULL,SET,
                           tagged,t_cc,t_i,new_deb,elements);


  if (result == ERROR)   /*   si erreur => imprimer un message   */
      {
         verdict = td_TYPE;
         goto error_exit;
      }


/*   retour sans erreur   */
/*   ------------------   */

normal_exit :

  t_ddl ("analyser_ChildInformation",deb,fin);   /* décrémentation de     */
                                                 /* la profondeur d'appel */
  return (NOERROR);



/*   retour avec erreur   */
/*   ------------------   */

error_exit :

  t_dmsg (opt,deb,fin,"analyser_ChildInformation",verdict);
                                                    /* message d'erreur*/

  t_ddl ("analyser_ChildInformation",deb,fin);  /* décrémentation de     */
                                                /* la profondeur d'appel */
  return (ERROR);

}
```

```
/*-------------------------------------------------------------------------*/
/*        Primitive d'analyse pour le type Date                            */
/*-------------------------------------------------------------------------*/

analyser_Date (opt,deb,fin,tagged,t_cc,t_i,new_deb,x)

  int opt;              /* optionel ?                */
  ptv_Byte deb,fin;     /* délimiteurs de l'ensemble */
  int tagged;           /* tag supplémentaire ?      */
  int t_cc;             /* classe                    */
  long t_i;             /* identificateur            */

  ptv_Byte *new_deb;    /* délimiteur du sous-ensemble résultat */
  std_Date *x;          / structure de donnée résultat         */


{

  int verdict,result;   /*   retour pour appels de primitives       */

    /*   primitives d'analyse pour les éléments */

  int analyser_IA5String ();


/*   incrementation de la profondeur d'appel  de primitives*/
/*   ------------------------------------------------------  */

  t_idl ("analyserDate",deb,fin);

    /*   appel de la macro-primitive pour les types de base élémentaires   */

  result = analyser_etmac (opt,deb,fin,tagged,t_cc,t_i,new_deb,x);


  if (result == ERROR)   /*   si erreur => imprimer un message   */
     {
        verdict = td_TYPE;
        goto error_exit;
     }


/*   retour sans erreur   */
/*   -----------------    */

normal_exit :

  t_ddl ("analyser_Date",deb,fin);   /* décrémentation de     */
                                     /* la profondeur d'appel */
  return (NOERROR);
```

```
/*   retour avec erreur   */
/*   ------------------   */

error_exit :

  t_dmsg (opt,deb,fin,"analyser_Date",verdict);
                                                /* message d'erreur*/

  t_ddl ("analyser_Date",deb,fin);  /* décrémentation de     */
                                    /* la profondeur d'appel */
  return (ERROR);

}
```

Annexe C : Spécifications LEX et YACC

Les spécifications qui suivent sont celles utilisées pour générer la primitive d'analyse lexicographique,à l'aide de LEX, et la primitive d'analyse syntaxique par le biais de YACC.

```
%{
/*-------------------------------------------------------------*/
/*                                                             */
/*                  Specification pour LEX                     */
/*-------------------------------------------------------------*/

#include "types.h"
#include "ytab.h"
#include "defines.h"
#include "externals.h"

%}


%%

%{
   /*  eliminer les blancs  */
   /*  -------------------  */
%}

[ \t\n]         a_fecho ();

%{
   /*  mots-cles de definition  */
   /*  -----------------------  */
%}

::=             {
                  a_fecho ();
                  return (t_EQUAL);
                }
DEFINITIONS     {
                  a_fecho ();
                  return (t_DEFINITIONS);
                }
BEGIN           {
                  a_fecho ();
                  return (t_BEGIN);
                }
END             {
                  a_fecho ();
                  return (t_END);
                }
```

```
%{
   /*   mots-cles pour le tagged type   */
   /*   ----------------------------   */
%}

UNIVERSAL         {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_UNIVERSAL);
                  }
APPLICATION       {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_APPLICATION);
                  }
PRIVATE           {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_PRIVATE);
                  }
IMPLICIT          {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_IMPLICIT);
                  }

%{
   /*   Types predefinis   */
   /*   ---------------    */
%}

BOOLEAN           {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_BOOLEAN);
                  }
INTEGER           {
                       /*  passage du numéro de ligne par le   */
                       /*  biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_INTEGER);
                  }
```

```
NULL            {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                  yylval.v_line = gv_line;
                  a_fecho ();
                  return (t_NULL);
                }

ANY             {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                  yylval.v_line = gv_line;
                  a_fecho ();
                  return (t_ANY);
                }

BIT[ \t\n]+STRING     {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                      yylval.v_line = gv_line;
                      a_fecho ();
                      return (t_BITSTRING);
                     }
OCTET[ \t\n]+STRING    {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                       yylval.v_line = gv_line;
                       a_fecho ();
                       return (t_OCTETSTRING);
                      }
SEQUENCE             {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                      yylval.v_line = gv_line;
                      a_fecho ();
                      return (t_SQ);
                     }

SEQUENCE[ \t\n]+OF   {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                      yylval.v_line = gv_line;
                      a_fecho ();
                      return (t_SQO);
                     }
SET                  {
                      /*  passage du numéro de ligne par le   */
                      /*  biais de yyval                      */

                      yylval.v_line = gv_line;
                      a_fecho ();
                      return (t_ST);
                     }
```

```
SET[ \t\n]+OF       {
                      /* passage du numéro de ligne par le   */
                      /* biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_STO);
                   }

CHOICE              {
                      /* passage du numéro de ligne par le   */
                      /* biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_CHOICE);
                   }

OPTIONAL            {
                      /* passage du numéro de ligne par le   */
                      /* biais de yyval                      */

                    yylval.v_line = gv_line;
                    a_fecho ();
                    return (t_OPTIONAL);
                   }

DEFAULT             {
                      /* passage du numéro de ligne par le   */
                      /* biais de yyval                      */

                   yylval.v_line = gv_line;
                   a_fecho ();
                   return (t_DEFAULT);
                  }
%{
   /*  Identificateurs de type et de nom de reference   */
   /*  ----------------------------------------------   */
%}

[A-Z][A-Za-z0-9_]*   {
                       /* passage du numéro de ligne et de   */
                       /* l'identificateur  reconnu par le   */
                       /* biais de yyval                     */

                      yylval.v_txt.t_line = gv_line;
                      yylval.v_txt.t_value = a_saves (yytext);
                      a_fecho ();
                      return (t_TIDENTIFIER);
                     }
```

```
[a-z][A-Za-z0-9_]$   {
                       /$  passage du numéro de ligne et de   $/
                       /$  l'identificateur  reconnu par le   $/
                       /$  biais de yyval                     $/

                       yylval.v_txt.t_line = gv_line;
                       yylval.v_txt.t_value = a_saves (yytext);
                       a_fecho ();
                       return (t_OIDENTIFIER);
                     }

%{
   /$  constantes numeriques et string   $/
   /$  ------------------------------    $/
%}

"-"[0-9][0-9]$      {
                       /$  passage du numéro de ligne et de   $/
                       /$  la constante reconnue par le       $/
                       /$  biais de yyval                     $/

                       yylval.v_txt.t_line = gv_line;
                       yylval.v_txt.t_value = a_saves (yytext);
                       a_fecho ();
                       return (t_NNUMBER);
                     }

[0-9][0-9]$         {
                       /$  passage du numéro de ligne et de   $/
                       /$  la constante reconnue par le       $/
                       /$  biais de yyval                     $/

                      yylval.v_txt.t_line = gv_line;
                      yylval.v_txt.t_value = a_saves (yytext);
                      a_fecho ();
                      return (t_UNUMBER);
                     }
"["      {
           a_fecho ();
           return (t_LB);
         }
"]"      {
           a_fecho ();
           return (t_RB);
         }
"{"      {
           a_fecho ();
           return (t_LCB);
         }
"}"      {
           a_fecho ();
           return (t_RCB);
         }
","      {
           a_fecho ();
           return (t_COMMA);
         }
```

```
"("     {
          a_fecho ();
          return (t_LP);
        }
")"     {
          a_fecho ();
          return (t_RP);
        }

%{
   /*  suppression des autres caracteres  */
   /*  ---------------------------------  */

     /*   émission d'un message d'erreur pour ceux-ci  */

%}

"-"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
":"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"+"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"="     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"_"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"§"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"&"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"^"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"%"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"$"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"£"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"@"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"!"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"~"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
```

```
"`"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"¦"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"\\"    {
          e_lmsg ("Unknown character(s) : discarded");
        }
"'"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
\"      {
          e_lmsg ("Unknown character(s) : discarded");
        }
";"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"."     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"?"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"/"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
"<"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
">"     {
          e_lmsg ("Unknown character(s) : discarded");
        }
```

```
/*-------------------------------------------------------------*/
/*                                                             */
/*                Spécification  YACC                          */
/*                                                             */
/*-------------------------------------------------------------*/

%{

#define YYDEBUG
#include "types.h"
#include "defines.h"
#include "externals.h"

%}

/*-------------------------------------------------------------*/
/*          Stack des valeurs echangees entre LEX et YACC      */
/*-------------------------------------------------------------*/

%union
   {
     LINE      v_line;
     s_txt     v_txt;
     s_int     v_int;
     s_long    v_long;
     s_tagged *v_tagged;
     s_liv     *v_liv;
     s_etbc    *v_etbc;
     s_tbe     *v_tbe;
     s_tb      *v_tb;
     s_dt      *v_dt;
     s_graphe  *v_graphe;
     s_ldt     *v_ldt;
     s_ldd     *v_ldd;
   }

/*-------------------------------------------------------------*/
/*              Definition des jetons fournis par LEX          */
/*-------------------------------------------------------------*/

%token           t_EQUAL,
                 t_BEGIN,
                 t_END,
                 t_DEFINITIONS
%token  <v_line> t_UNIVERSAL,
                 t_APPLICATION,
                 t_PRIVATE,
                 t_IMPLICIT,
%token           t_LP,
                 t_RP,
                 t_LB,
                 t_RB,
                 t_LCB,
                 t_RCB,
                 t_COMMA,
%token  <v_line> t_BOOLEAN,
                 t_INTEGER,
                 t_BITSTRING,
                 t_OCTETSTRING,
                 t_NULL,
```

```
                t_ANY,
                t_SQ,
                t_SQO,
                t_ST,
                t_STO,
                t_CHOICE,
                t_OPTIONAL,
                t_DEFAULT

%token  <v_txt>  t_TIDENTIFIER,
                 t_OIDENTIFIER,
                 t_NNUMBER,
                 t_UNUMBER


/*-------------------------------------------------------------*/
/*  Definition des types des valeurs retournees par les regles */
/*-------------------------------------------------------------*/

%type  <v_etbc>   AlternativeTypeList
%type  <v_int>    Class
%type  <v_tb>     ConstructedType
%type  <v_tbe>    ElementaryType
%type  <v_etbc>   ElementTypes
%type  <v_liv>    NamedBitList
%type  <v_liv>    NamedBit
%type  <v_liv>    NamedNumberList
%type  <v_liv>    NamedNumber
%type  <v_etbc>   NamedType
%type  <v_int>    NumericValue
%type  <v_etbc>   OptionalType
%type  <v_etbc>   OptionalTypeList
%type  <v_int>    PositiveNumericValue
%type  <v_dt>     TaggedType
%type  <v_tagged> Tag
%type  <v_tb>     Type


/*-------------------------------------------------------------*/
/*                Definition de la grammaire X409              */
/*-------------------------------------------------------------*/


%start ModuleDefinition


%%

/*-------------------------------------------------------------*/
/*                        Module                               */
/*-------------------------------------------------------------*/
```

```
ModuleDefinition   :


    t_TIDENTIFIER
    t_DEFINITIONS
    t_EQUAL
    t_BEGIN
    ModuleBody
    t_END
      {
        gv_module = $<v_txt>1.t_value;
      }
  |
    t_TIDENTIFIER
    error
      {
        e_ymsg ("DEFINITIONS should occur right after Name");
      }

  |
    t_TIDENTIFIER
    t_DEFINITIONS
    error
      {
        e_ymsg ("::= should occur right after DEFINITIONS");
      }
  |
    t_TIDENTIFIER
    t_DEFINITIONS
    t_EQUAL
    error
      {
        e_ymsg ("BEGIN or END missing");
      }

  ;   /*   returns nothing   */

ModuleBody :

    empty
  |
    DefinitionList

  ;   /*   returns nothing   */

DefinitionList :

    Definition
  |
    DefinitionList
    Definition

  ;   /*   returns nothing   */

Definition :

    TypeDefinition

  ;   /*   returns nothing   */
```

```
/*-------------------------------------------------------------*/
/*                  Definition de type                         */
/*-------------------------------------------------------------*/

TypeDefinition :

    t_TIDENTIFIER
    t_EQUAL
    TaggedType
      {
        a_adt ($<v_txt>1,$<v_dt>3);
      }
  |
    t_TIDENTIFIER
    error
      {
        e_ymsg ("::= should occur right after Type identifier");
      }
  |
    t_TIDENTIFIER
    t_EQUAL
    error
      {
        e_ymsg ("Not a valid tagged type following ::=");
      }


  ;   /*  returns nothing  */

/*-------------------------------------------------------------*/
/*                      Tagged type                            */
/*-------------------------------------------------------------*/

TaggedType :

    Type
      {
        $<v_dt>$ = a_cdt (NULL,$<v_tb>1);
      }
  |
    Tag
    t_IMPLICIT
    Type
      {
        $<v_tagged>1->tag_line = $<v_line>2;
        $<v_tagged>1->tagged   = IMPLICIT;
        $<v_dt>$ = a_cdt ($<v_tagged>1,$<v_tb>3);
      }
  |
    Tag
    t_IMPLICIT
    error
      {
        e_ymsg ("Not a valid type specification following IMPLICIT");
      }
```

```
  |
    Tag
    error
      {
        e_ymsg ("Not a valid type specification following tag");
      }
  |
    Tag
    Type
      {
        $(v_tagged)1->tag_line = UNKNOWN;
        $(v_tagged)1->tagged = EXPLICIT;
        $(v_dt)$ = a_cdt ($(v_tagged)1,$(v_tb)2);
      }

  ;

Tag  :

    t_LB
    Class
    t_UNUMBER
    t_RB
      {
        $(v_tagged)$ = a_ctag ($(v_int)2,$(v_txt)3);
      }

  |
    t_LB
    Class
    error
      {
        e_ymsg ("Not a valid IDCode");
      }
  |
    t_LB
    Class
    t_UNUMBER
    error
      {
        e_ymsg ("] missing");
      }

 ;   /*   returns (s_tagged *)   */

Class :

    empty
      {
        $(v_int)$.i_line = gv_line;
        $(v_int)$.i_value = CONTEXT;
      }
  |
    t_UNIVERSAL
      {
        $(v_int)$.i_line = $(v_line)1;
        $(v_int)$.i_value = UNIVERSAL;
      }
```

```
 |
   t_APPLICATION
     {
       $(v_int)$.i_line = $(v_line)1;
       $(v_int)$.i_value = APPLICATION;
     }
 |
   t_PRIVATE
     {
       $(v_int)$.i_line = $(v_line)1;
       $(v_int)$.i_value = PRIVATE;
     }

 ;   /*   returns (s_int)   */


/*-------------------------------------------------------------*/
/*                          Type                               */
/*-------------------------------------------------------------*/

Type :

   ElementaryType
     {
       $(v_tb)$ = a_ctbe ($(v_tbe)1);
     }
 |
   ConstructedType
     {
       $(v_tb)$ = $(v_tb)1;
     }
 ;

/*-------------------------------------------------------------*/
/*                      Type elementaire                       */
/*-------------------------------------------------------------*/

ElementaryType :

   t_BOOLEAN
     {
       $(v_tbe)$ = a_cte ("BOOLEAN",$(v_line)1,YES,NULL);
     }

 |
   t_INTEGER
     {
       $(v_tbe)$ = a_cte ("INTEGER",$(v_line)1,YES,NULL);
     }

 |
   t_INTEGER
   t_LCB
   NamedNumberList
   t_RCB
     {
       $(v_tbe)$ = a_cte ("INTEGER",$(v_line)1,
                          YES,$(v_liv)3);
     }
```

```
!
  t_INTEGER
  t_LCB
  error
    {
      e_ymsg
          ("reference name for named number expected after (");
    }
!
  t_INTEGER
  t_LCB
  NamedNumberList
  error
    {
      e_ymsg (") expected to close the named number list");
    }

!
  t_BITSTRING
    {
      $(v_tbe)$ = a_cte ("BIT STRING",$(v_line)1,YES,NULL);
    }

!
  t_BITSTRING
  t_LCB
  NamedBitList
  t_RCB
    {
      $(v_tbe)$ = a_cte ("BIT STRING",$(v_line)1,
                    YES,$(v_liv)3);
    }

!
  t_BITSTRING
  t_LCB
  error
    {
      e_ymsg ("reference name for named bit expected after (");
    }

!
  t_BITSTRING
  t_LCB
  NamedBitList
  error
    {
      e_ymsg (") expected to close the named bit list");
    }

!
  t_OCTETSTRING
    {
      $(v_tbe)$ = a_cte ("OCTET STRING",$(v_line)1,
                    YES,NULL);
    }
```

```
  |
    t_NULL
      {
        $(v_tbe)$ = a_cte ("NULL",$(v_line)1,
                        YES,NULL);
      }

  |
    t_ANY
      {
        $(v_tbe)$ = a_cte ("ANY",$(v_line)1,
                        YES,NULL);
      }

  |
    t_TIDENTIFIER
      {
        $(v_tbe)$ = a_cte ($(v_txt)1.t_value,
                                $(v_txt)1.t_line,NO,NULL);
      }

  ;   /*   returns (s_tbe *)   */

NamedNumberList :

    NamedNumber
      {
        $(v_liv)$ = a_piv ($(v_liv)1);
      }

  |
    NamedNumberList
    t_COMMA
    NamedNumber
      {
        $(v_liv)$ = a_aiv ($(v_liv)1,$(v_liv)3);
      }

  |
    NamedNumberList
    t_COMMA
    error
      {
        e_ymsg
           ("reference name for named
                  number expected after COMMA (,)");
      }

  ;   /*   returns (s_liv *)   */


NamedNumber :

    t_OIDENTIFIER
    t_LP
    NumericValue
    t_RP
      {
        $(v_liv)$ = a_civ ($(v_txt)1,$(v_txt)3);
      }
```

```
  |
    t_OIDENTIFIER
    error
      {
        e_ymsg ("( expected after
                    reference name for named number");
      }

  |
    t_OIDENTIFIER
    t_LP
    error
      {
        e_ymsg ("number expected after (");
      }

  |
    t_OIDENTIFIER
    t_LP
    NumericValue
    error
      {
        e_ymsg (") expected after number");
      }

  ;   /*   returns (s_liv *)   */


NumericValue :

    t_NNUMBER
      {
        $(v_txt)$.t_line = $(v_txt)1.t_line;
        $(v_txt)$.t_value = $(v_txt)1.t_value;
      }

  |
    t_UNUMBER
      {
        $(v_txt)$.t_line = $(v_txt)1.t_line;
        $(v_txt)$.t_value = $(v_txt)1.t_value;
      }

  ;   /*   returns (s_txt)   */

NamedBitList :

    NamedBit
      {
        $(v_liv)$ = a_piv ($(v_liv)1);
      }

  |
    NamedBitList
    t_COMMA
    NamedBit
      {
        $(v_liv)$ = a_aiv ($(v_liv)1,$(v_liv)3);
      }
```

```
  |
    NamedBitList
    t_COMMA
    error
      {
        e_ymsg ("reference name
                  for named bit expected after COMMA (,)");
      }

  ;   /*   returns (s_liv *)   */

NamedBit :

    t_OIDENTIFIER
    t_LP
    PositiveNumericValue
    t_RP
      {
        $<v_liv>$ = a_civ ($<v_txt>1,$<v_txt>3);
      }
  |
    t_OIDENTIFIER
    error
      {
        e_ymsg ("( expected after reference name for named bit");
      }

  |
    t_OIDENTIFIER
    t_LP
    error
      {
        e_ymsg ("unsigned number expected after (");
      }
  |
    t_OIDENTIFIER
    t_LP
    PositiveNumericValue
    error
      {
        e_ymsg (") expected after unsigned number");
      }
  ;   /*   returns (s_liv *)   */

PositiveNumericValue :

    t_UNUMBER
      {
        $<v_txt>$.t_line = $<v_txt>1.t_line;
        $<v_txt>$.t_value = $<v_txt>1.t_value;
      }
  ;   /*   returns (s_txt)   */
```

```
/*-------------------------------------------------------------*/
/*                    Type constructeur                        */
/*-------------------------------------------------------------*/

ConstructedType :

    t_SQ
      {
        $(v_tb)$ = a_csqoany ($(v_line)1);
      }
  |
    t_SQO
    TaggedType
      {
        $(v_tb)$ = a_csqo ($(v_line)1,$(v_dt)2);
      }
  |
    t_SQ
    t_LCB
    ElementTypes
    t_RCB
      {
        $(v_tb)$ = a_csq ($(v_line)1,$(v_etbc)3);
      }

  |
    t_SQ
    t_LCB
    ElementTypes
    error
      {
        e_ymsg (") expected after
                the specification of sequence elements");
      }
  |
    t_ST
      {
        $(v_tb)$ = a_cstoany ($(v_line)1);
      }
  |
    t_STO
    TaggedType
      {
        $(v_tb)$ = a_csto ($(v_line)1,$(v_dt)2);
      }
  |
    t_ST
    t_LCB
    ElementTypes
    t_RCB
      {
        $(v_tb)$ = a_cst ($(v_line)1,$(v_etbc)3);
      }
```

```
  |
    t_CHOICE
    t_LCB
    AlternativeTypeList
    t_RCB
      {
        $<v_tb>$ = a_cchoice ($<v_line>1,$<v_etbc>3);
      }

  ;   /*   returns (s_tb *)   */

ElementTypes :

    empty
      {
        $<v_etbc>$ = NULL;
      }

  |

    OptionalTypeList
      {
        $<v_etbc>$ = $<v_etbc>1;
      }

  ;   /*   returns (s_etbc *)   */

OptionalTypeList :

    OptionalType
      {
        $<v_etbc>$ = a_pto ($<v_etbc>1);
      }

  |
    OptionalTypeList
    t_COMMA
    OptionalType
      {
        $<v_etbc>$ = a_ato ($<v_etbc>1,$<v_etbc>3);
      }

  |
    OptionalTypeList
    t_COMMA
    error
      {
        e_ymsg ("specification of
                element type expected  after COMMA (,)");
      }

  ;   /*   returns (s_etbc *)   */

AlternativeTypeList :

    NamedType
      {
        $<v_etbc>$ = a_pa ($<v_etbc>1);
      }
```

```
  |
    AlternativeTypeList
    t_COMMA
    NamedType
      {
        $<v_etbc>$ = a_aa ($<v_etbc>1,$<v_etbc>3);
      }
  ;   /*   returns (s_etbc *)   */

OptionalType :

    NamedType
      {
        $<v_etbc>$ = a_cto ($<v_etbc>1,NO,NULL);
      }

  |

    NamedType
    t_OPTIONAL
      {
        $<v_etbc>$ = a_cto ($<v_etbc>1,YES,NULL);
      }

  ;   /*   returns (s_etbc *)   */

NamedType :

    t_OIDENTIFIER
    TaggedType
      {
        $<v_etbc>$ = a_ctn ($<v_txt>1,$<v_dt>2);
      }
  |
    t_OIDENTIFIER
    error
      {
        e_ymsg ("invalid type
                 specification after reference name");
      }
  |
    TaggedType
      {
        $<v_etbc>$ = a_ctnsn ($<v_dt>1);
      }

  ;   /*   returns (s_etbc *)   */



empty : ;
```